Universidad Complutense de Madrid

Facultad de Informática

Co-simulación HW/SW en Raspberry Pi

Alumno **Miguel Higuera Romero**

Director de proyecto **José Luis Risco Martín**

Trabajo de Fin de Grado

Grado en Ingeniería de Computadores

Junio de 2016

Índice general

Palabras clave

Palabras clave en Español

- Modelado y simulación
- Sistemas de eventos discretos
- Formalismo DEVS y xDEVS
- Raspberry Pi
- Módulos del kernel de Linux
- Circuitos lógicos y secuenciales

Keywords in English

- Modeling and simulation
- Discrete events system
- DEVS formalism and xDEVS
- Raspberry Pi
- Linux kernel Modules
- Logic and secuential circuits

Resumen

Resumen en Español

En el mundo de la simulación existen varios tipos de sistemas reales, entre los que se encuentran los sistemas de eventos discretos. Para poder simular estos sistemas se pueden utilizar, entre otras, herramientas basadas en el formalismo DEVS (Discrete EVents system Specification), como la utilizada en este proyecto: xDEVS.

La simulación posee una importancia muy elevada en campos como la educación y la ciencia, y en ocasiones es necesario incluir datos del medio físico o sacar información al exterior del simulador. Por ello es necesario contar con herramientas que puedan realizar simulaciones utilizando sensores, actuadores, circuitos externos, etc., o lo que es lo mismo, que puedan realizar co-simulaciones entre software y hardware. De esta forma se puede facilitar el desarrollo de sistemas por medio de modelado y simulación, pudiendo extraer el hardware gradualmente y analizar los resultados en cada etapa.

Este proyecto es de carácter incremental, y trata de extender la funcionalidad de la plataforma xDEVS para poder realizar co-simulaciones entre hardware y software sobre una Raspberry Pi. Para ello se van a utilizar circuitos lógicos como hardware externo y se enlazarán al simulador a través de ficheros de dispositivo, gestionados por módulos del kernel de Linux. Como caso de estudio se desarrolla la co-simulación entre hardware y software completa de un ascensor de siete plantas para mostrar el uso y funcionamiento en xDEVS, extrayendo los circuitos integrados de uno en uno.

Sumary in English

In the world of simulation there are several types of real systems, among which are the discrete event systems. These systems can be simulated using tools based on the DEVS formalism (Discrete EVents system Specification), like xDEVS, which is the platform used in this paper.

The simulation has a high relevance in fields like education and science, and sometimes it is necessary to include data from a physical environment or to send information outside the simulator. Thus, it is necessary to have tools that are able to run simulations using sensors, actuators, external circuits, etc. These tools should be able to run co-simulations between software and hardware. Thereby the development of systems through modeling and simulation can be improved, with the possibility of incrementally including hardware to the external circuit and analyzing the results at different stages.

This is an incremental project, which tries to extend the functionality of the xDEVS platform in order to run co-simulations between hardware and software on a Raspberry Pi. For this purpose

logic circuits are used as external hardware. They will be linked to the simulator through device files managed by Linux kernel modules. As a case study, a complete co-simulation between hardware and software of a seven-floor elevator is performed to illustrate the use and functioning in xDEVS, extracting integrated circuits one by one.

Capítulo 1

Introducción

La simulación es la reproducción de un fenómeno real mediante un modelo más sencillo, con mayor aptitud para ser estudiado. El fin de la simulación no es otro que comprender el funcionamiento del sistema real o evaluar distintas experiencias con el mismo. Además de obtener los resultados o salidas del sistema real, el modelo puede revelar información necesaria que en la realidad está oculta. Por ello la simulación es de gran utilidad en muchos campos como la educación, la ciencia o la informática.

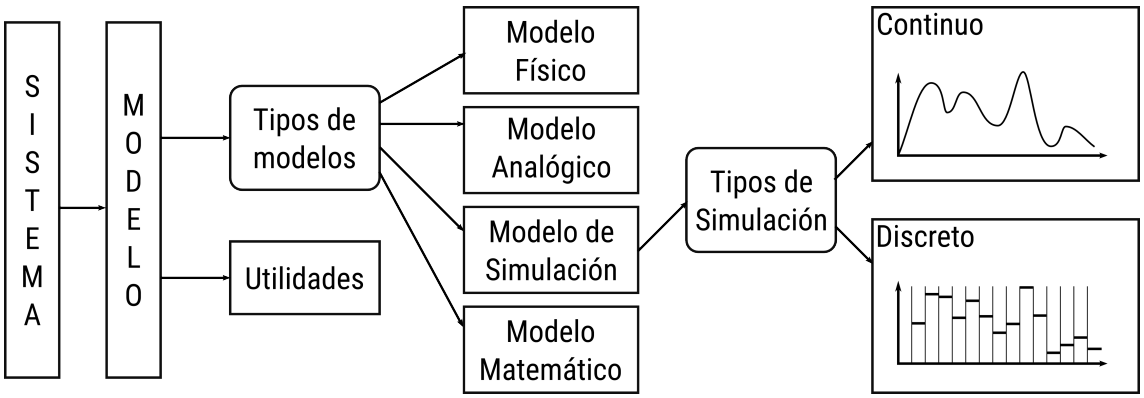

Figura 1.1: Modelos y Simulación

En ocasiones, es de vital importancia incluir ciertas medidas u operaciones especiales en la simulación que sólo pueden llevarse a cabo por medio de hardware o, al contrario, comprobar el correcto funcionamiento del hardware incorporándolo en un entorno simulado. Por ejemplo, en un simulador de vuelo se necesita transmitir al simulador las órdenes que ejecuta el piloto, las cuales son tomadas del medio físico a través de sensores.

La Figura 1.1 muestra que, partiendo de un sistema real, se puede elaborar un modelo que nos dé información del sistema al someterlo a eventos concretos. Los modelos pueden ser de varios tipos: físicos, analógicos, matemáticos o de simulación. Los modelos de simulación pueden representar sistemas de eventos continuos o discretos. Los últimos, los sistemas de eventos discretos, son los que se van a cubrir en este documento.

1.1. Antecedentes

Hoy en día, emerge la era del Internet de las Cosas (IoT, del inglés Internet Of Things), haciendo que el ser humano forme parte activa de los sistemas como un componente más. Por lo tanto, se necesita un alto nivel de calidad, precisión y eficiencia en los sistemas a desarrollar. A su vez, los sistemas del IoT utilizan típicamente sistemas heterogéneos, que combinan hardware y software en los modelos de todo un ecosistema. Por ejemplo coches inteligentes, casas inteligentes o edificios inteligentes en entornos integrados de energía.

Por otro lado el diseño, desarrollo e implementación de sistemas empotrados en tiempo real es un reto hoy en día a nivel de ingeniería de sistemas. Las restricciones en la gestión del tiempo real, los plazos de ejecución de tareas, etc., plantea ciertos problemas en el diseño de estos dispositivos. En numerosas ocasiones, en ingeniería de sistemas, se reccurre a técnicas de modelado y simulación para el desarrollo de sistemas. La construcción de un modelo virtual permite realizar un análisis adecuado mediante la simulación, al mismo tiempo que reduce el esfuerzo, los costes y los riesgos, y permite mejorar la calidad y las capacidades del sistema en desarrollo. Sin embargo, la mayoría de las técnicas de modelado y simulación no permiten un diseño incremental que permita incorporar componentes hardware gradualmente, sino que se descarta el modelo virtual en las etapas avanzadas del desarrollo.

1.2. Objetivos

Este proyecto es de carácter incremental y trata de extender la herramienta de simulación xDEVS, basada en el formalismo DEVS, para que permita realizar co-simulaciones entre hardware y software. Por lo tanto, el objetivo no es tanto conseguir eficiencia en la comunicación con el hardware o los drivers utilizados, sino conseguir funcionalidad básica para que sirva como punto de partida en proyectos o trabajos futuros.

Al incluir hardware en el simulador se busca abrir las puertas a nuevos modelos que puedan tratar sensores, actuadores o incluso artefactos robóticos que realicen tareas complejas en un entorno simulado. También se pretende facilitar el diseño de sistemas aportando la posibilidad de utilizar modelado y simulación a lo largo de todo el desarrollo. Todo esto bajo la imposición de no modificar las especificaciones del formalismo DEVS.

El fin de este trabajo es elaborar el procedimiento a seguir para poder realizar una cosimulación entre hardware y software, demostrar el funcionamiento y elaborar un caso de estudio concreto. Para ello se va a elaborar la co-simulación HW/SW de un Sumador sencillo, con el fin de mostrar el funcionamiento, y como caso de estudio se va a desarrollar el circuito externo del controlador de un ascensor de siete plantas, partiendo de la simulación de todos sus componentes y extrayéndolos uno a uno al circuito externo.

1.3. Plan de trabajo

La organización de este trabajo ha consistido en reuniones periódicas en las que el director del proyecto y el alumno se han puesto al día de los avances. El tránsito general ha comprendido los siguientes campos:

- Estudio del formalismo DEVS y de la plataforma xDEVS

- Estudio y puesta a punto de la Raspberry Pi 2 B para una correcta co-simulación entre hardware y software.
- Estudio sobre drivers y módulos del kernel de GNU/Linux.
- Estudio y pruebas de comunicación entre circuitos externos y Raspberry Pi.
- Modelado y co-simulación HW/SW completa de sistemas sencillos en xDEVS.
- Elaboración del caso de estudio. Modelado y co-simulación del sistema de un ascensor.
- Estudio de edición de textos científicos con LaTeX para la elaboración de esta memoria.

Capítulo 2

Raspberry Pi

2.1. Introducción

Raspberry Pi es un ordenador de placa reducida y bajo coste, que fue desarrollado en el Reino Unido con el fin de integrar las ciencias de la computación en la educación de las escuelas. El proyecto lo desarrolla la fundación *Raspberry Pi* y su primer lanzamiento fue la Raspberry Pi Modelo A en Febrero de 2012. la cual se muestra en la Figura 2.1. Los productos que desarrolla son de uso libre pero con propiedad registrada, de forma que la fundación mantiene el control de la plataforma pero permitiendo libertad de uso tanto a nivel educativo como particular.

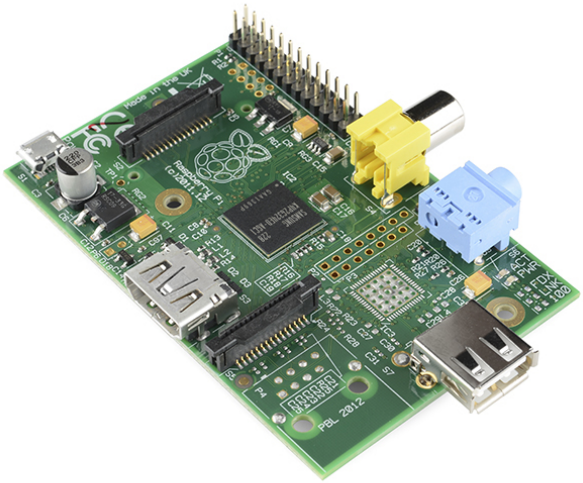

Figura 2.1: Raspberry Pi Model A - Imagen de SparkFun Electronics - CC BY 2.0

El diseño hardware contiene un SoC *Broadcom BCM2835*, que integra:

- Un procesador *ARM1176JZF-S* a 700MHz.
- Un procesador gráfico *VideoCore IV*.
- 512 MB de memoria RAM.

Además, carece de disco duro o unidad de estado sólido; en su lugar utiliza una tarjeta SD para el almacenamiento, tanto del sistema operativo como de los ficheros personales.

La fundación *Raspberry Pi* da soporte para las descargas de las distribuciones GNU/Linux para la arquitectura ARM, como son *Raspbian*, *RISC OS 5*, *Arch Linux ARM* o *Pidora* entre los más destacados. Estas distribuciones y más pueden descargarse en `https://www.raspberrypi.org/downloads/`.

2.2. Raspberry Pi 2 Modelo B

Para el desarrollo de este proyecto se va a utilizar la *Raspberry Pi 2 Modelo B*, por contener conectividad de red Ethernet y un periférico GPIO con mejores posibilidades para el proyecto.

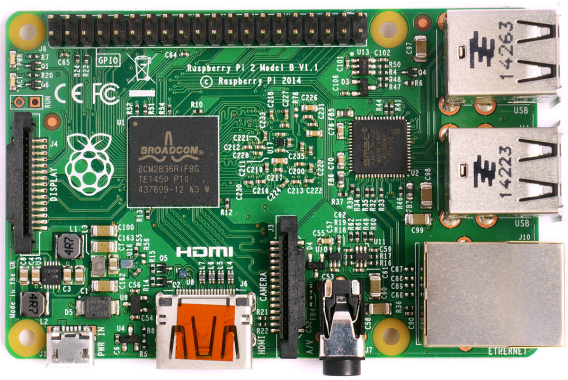

Figura 2.2: Raspberry Pi 2 Modelo B - Imagen De Multicherry - CC BY-SA 4.0

A diferencia de las primeras placas, la Raspberry Pi 2 B, contiene un SoC *Broadcom BCM2836* que integra:

- Un procesador quad-core *ARM Cortex A7* a 900 MHz
- Un procesador gráfico *VideoCore IV*.
- 1 GB de memoria RAM.

Para el almacenamiento dispone de una ranura MicroSD. En la tarjeta se instalará la distribución GNU/Linux llamada *Raspbian Jessie*, derivada de *Debian Jessie*, que dispone de la adaptación y optimización necesarias para procesadores ARM.

En la Figura 2.3 se muestran los pines del periférico GPIO, el cual se utilizará para las conexiones con el hardware externo. Se ha elegido este periférico por ser de propósito general y disponer de varios pines configurables para establecer una comunicación.

En la distribución de Raspbian ya se da un driver preinstalado para el uso del GPIO, por medio de comandos a ficheros de dispositivo; pero en este trabajo se hará uso de la librería del kernel `linux/gpio.h` para gestionar los pines, dentro de los módulos propios que se implementen.

En último lugar, únicamente mencionar que la Raspberry Pi utiliza 3.3V en los pines del GPIO. Algunos circuitos externos necesitan 5V para funcionar, por lo que será necesario un conversor de

PIN#	NAME		NAME	PIN#
01	3.3V		5V	02
03	GPIO02		5V	04
05	GPIO03		GND	06
07	GPIO04		GPIO14	08
09	GND		GPIO15	10
11	GPIO17		GPIO18	12
13	GPIO27		GND	14
15	GPIO22		GPIO23	16
17	3.3V		GPIO24	18
19	GPIO10		GND	20
21	GPIO09		GPIO25	22
23	GPIO11		GPIO08	24
25	GND		GPIO07	26
27	ID_SD		ID_SC	28
29	GPIO05		GND	30
31	GPIO06		GPIO12	32
33	GPIO13		GND	34
35	GPIO19		GPIO16	36
37	GPIO26		GPIO20	38
39	GND		GPIO21	40

Figura 2.3: Periférico GPIO y sus pines

3.3V a 5V para enviar señales al circuito externo y un conversor de 5V a 3.3V para recibir señales en la Raspberry. Si entraran 5V por algún pin del GPIO cabe la posibilidad de que se queme internamente la placa.

2.3. Configuración para la simulación

La Raspberry Pi con Raspbian instalado, por defecto, no nos permite la compilación de módulos del kernel. Por ello se ha recurrido al uso de *RPi Source*, un proyecto de Software Libre, alojado en https://github.com/notro/rpi-source, que nos permitirá, tras seguir unos pasos, poder compilar módulos. La instalación y uso de *RPi Source* se especifican en la wiki del proyecto, incluida en su espacio de Github.

Por otro lado, en Raspbian, hay que comprobar que el JRE de Java instalado sea el de *Oracle*, para evitar incompatibilidades con la plataforma xDEVS.

Capítulo 3

El formalismo DEVS

3.1. Introducción

DEVS es la abreviación de ***D**iscrete **EV**ents **S**ystem Specification*. Es un formalismo modular y jerárquico para el modelado y análisis de sistemas reales que puedan ser representados como sistemas de eventos discretos, sistemas de estado continuo o sistemas híbridos entre los dos primeros. Respectivamente son descritos por tablas de estado de transición, ecuaciones diferenciales o ambas. En cualquier caso es un sistema de eventos cronometrado.

El formalismo fue creado por Bernard P. Zeigler, profesor emérito de la Universidad de Arizona en EEUU, a mediados de los años 70. Se puede ver como una extensión del formalismo de la máquina de Moore, un autómata de estados finitos cuyas salidas están determinadas por el estado en el que se encuentra. De acuerdo a la teoría de DEVS, un sistema real se puede representar como un modelo construido en base a la experimentación que se quiera realizar y las condiciones de trabajo existentes. De esta forma, el modelo está restringido a la experimentación para la que fue hecho.

En el formalismo DEVS, un sistema está formado por subsistemas, los cuales se corresponden con modelos *atómicos* o modelos *acoplados*. Los últimos están compuestos a su vez por otros modelos atómicos y/o acoplados. De esta forma, al disponer de sistemas complejos, es fácil probar los subsistemas por separado.

Bajo un punto de vista general, un modelo DEVS está caracterizado por generar eventos de salida Y, en relación con el estado en el que se encuentre S y las entradas recibidas X, cada cierto tiempo.

Figura 3.1: Representación simple de un modelo DEVS

3.2. Descripción del formalismo DEVS

DEVS define el comportamiento de un sistema real, al mismo tiempo que su estructura interna. El comportamiento es descrito utilizando eventos de entrada y salida, y transiciones entre estados concretos. La estructura del sistema varía según su naturaleza.

En la Figura 3.2, como ejemplo intuitivo, se muestra el modelado en DEVS del juego del Ping-Pong, el cual será explicado detalladamente en las próximas secciones de este capítulo.

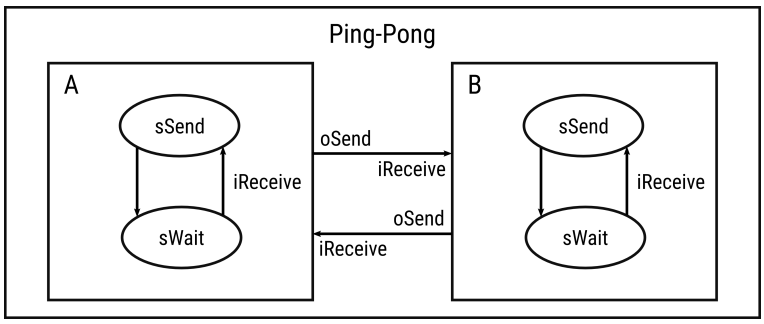

Figura 3.2: Modelado DEVS para el juego del Ping-Pong

En el PingPong hay dos jugadores, A y B, y ambos están caracterizados por lo siguiente:

- Entrada : *iReceive*. Revela si el adversario ha lanzado la pelota y hará que se pase al estado *sSend*.
- Salida : *oSend*. Indica si el jugador ha lanzado la pelota (1) o aún no lo ha hecho (0).
- Estados : *sSend* y *sWait*. Según las entradas y el tiempo transcurrido va cambiando de uno a otro.

3.2.1. Modelo DEVS atómico

Un modelo DEVS simple o atómico se representa bajo la 7-tupla:

$$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

Donde:

- X : Conjunto de pares $(puerto, valor)$ de los eventos de entrada.
- S : Conjunto de estados.
- Y : Conjunto de pares $(puerto, valor)$ de los eventos de salida.
- $\delta_{int} : S \to S$, función de transición interna.
- $\delta_{ext} : Q \times X \to S$, función de transición externa, con

$$Q = \{(s, t_e) | s \in S, t_e \in [0, \tau(s)]\}$$

- $\lambda : S \to Y$, función de salida.
- $\tau : S \to \mathbb{R}_0^+$, función de duración.

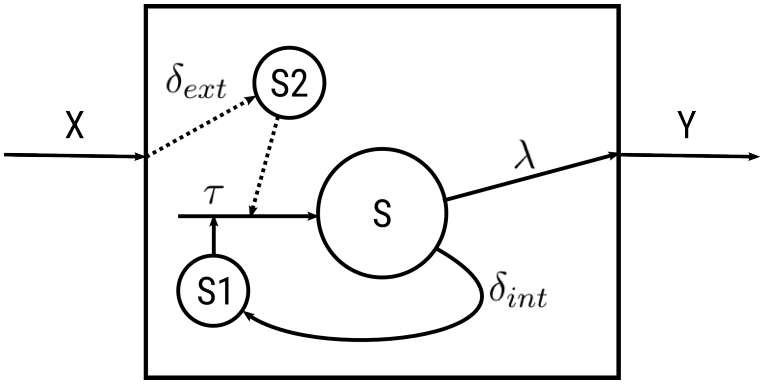

Figura 3.3: Modelo atómico DEVS

La Figura 3.3 muestra el comportamiento interno de un modelo DEVS atómico. Cada atómico tiene unas entradas X y unas salidas Y que van a permitirle comunicarse con otros modelos. La función de duración τ define el tiempo de vida del estado actual del modelo S. Los estados se actualizan al pasar el tiempo con la función de transición interna $\delta_{int} : S_1 \rightarrow S$. Al actualizarse el estado, se reflejan los cambios en las salidas Y por medio de la función de salida λ. En cualquier momento, el modelo, puede recibir eventos de entrada que actualizarán el estado del modelo por medio de la función de transición externa $\delta_{ext} : S_2 \rightarrow S$, y reiniciarán el tiempo de vida del estado.

A continuación se muestra la descripción según el formalismo DEVS de los modelos atómicos A y B del juego de Ping-Pong, cuya representación gráfica se puede ver en la Figura 3.4 y el diagrama temporal en la Figura 3.5.

$$Jugador = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

- $X = \{(in, iReceive)\}$, con $iReceive \in \{0, 1\}$
- $S = (phase, \sigma, iReceive)$, con $phase \in \{"sWait", "sSend"\}, \sigma \in \mathbb{R}_0^+$
- $Y = \{(out, oSend)\}$, con $oSend \in \{0, 1\}$
- $\delta_{int}("sSend", \sigma, iReceive) = ("sWait", \infty, \emptyset)$
- $\delta_{ext}("sWait", \sigma, iReceive, t_e, (in, iReceive')) = ("sSend", t_{iReceive}, iReceive')$
- $\lambda("sSend", \sigma, iReceive) = 1$
  $\lambda("sWait", \sigma, iReceive) = 0$

3.2.2. Modelo DEVS acoplado

Un modelo acoplado DEVS se representa bajo la 6-tupla:

$$CM = \langle X, Y, C_i, EIC, IC, EOC \rangle$$

Donde:

- X : Conjunto de pares $(puerto, valor)$ de los eventos de entrada.

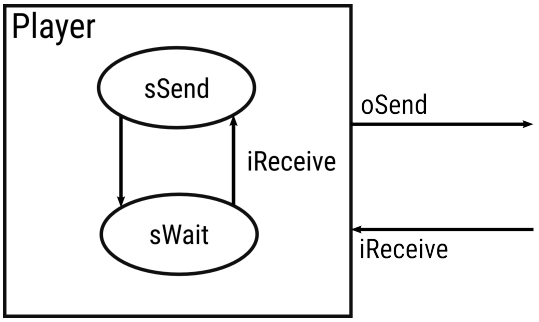

Figura 3.4: Modelo atómico DEVS de un jugador del PingPong

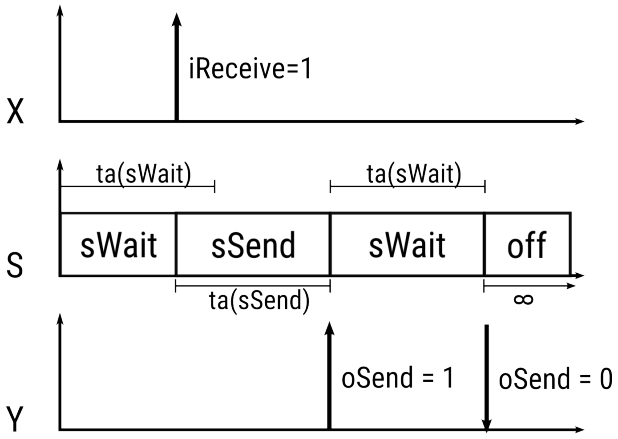

Figura 3.5: Diagrama de tiempo del modelo atómico DEVS de un jugador del PingPong

- Y : Conjunto de pares $(puerto, valor)$ de los eventos de salida.
- C_i : Conjunto de componentes de cada modelo i.
- EIC : Función de acoplamiento de las entradas del modelo acoplado a las entradas de C_i.
- IC : Función de traslación de las salidas de C_i a las entradas de C_j.
- EOC : Función de acoplamiento de las salidas de C_i a las salidas del modelo acoplado.

El modelo acoplado DEVS está compuesto por modelos atómicos y/o acoplados cuyas entradas y salidas han sido interconectadas entre ellos y el acoplado que los contiene. Cada modelo interno tiene un identificador i y un conjunto de componentes C_i. Las funciones de traslación EIC, IC y EOC conectan, respectivamente, las entradas del acoplado con las entradas de C_i, las salidas de C_i con las entradas de C_j y las salidas de C_i con las salidas del acoplado.

Ya definido el modelo atómico del jugador del Ping-Pong, se describe a continuación, según el formalismo DEVS, el modelo acoplado del juego, mostrado en la Figura 3.2.

$$PingPong = \langle X, Y, C_i, EIC, IC, EOC \rangle$$

- $X = \emptyset$
- $Y = \emptyset$

- $C_{Jugador} : \{A, B\}$
- $EIC = \emptyset$
- $IC = \{(A.out, B.in), (B.out, A.in)\}$
- $EOC = \emptyset$

3.2.3. Simulación de modelos DEVS

Los modelos DEVS pueden ser simulados de manera eficiente y sencilla. El algoritmo básico de simulación es el siguiente:

1 - Encontrar el modelo atómico de C_i que, según su función τ y el tiempo trascurrido t, deba ser el próximo en ejecutar su δ_{int}.

2 - Se adelanta t hasta t_n, siendo t_n el tiempo de la transición δ_{int}.

3 - Se ejecutan las funciones λ y δ_{int} del modelo atómico de C_i.

4 - Tras ejecutar la función λ se propagan las nuevas salidas Y a todos los atómicos conectados con ellas y se ejecutan las funciones δ_{ext} de los mismos.

5 - Se vuelve al paso 1.

Para la implementación de este algoritmo en un lenguaje de programación debe tenerse en cuenta que cada modelo atómico tiene asociado un simulador DEVS y cada modelo acoplado un coordinador DEVS, como se puede ver en la Figura 3.6.

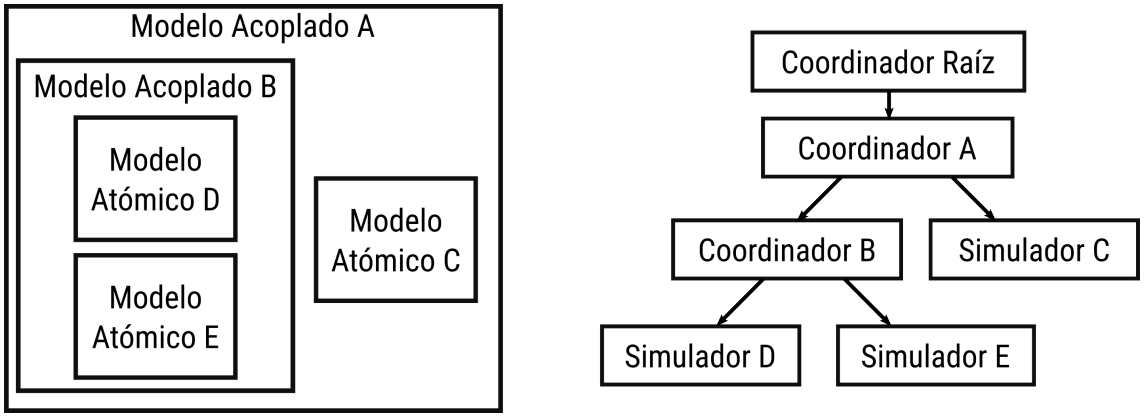

Figura 3.6: Estructura jerárquica de la simulación DEVS

Actualmente existen diversos proyectos que permiten desarrollar modelos DEVS y simularlos, por ejemplo:

- DEVSJAVA
- DEVS/C++
- PowerDEVS

- xDEVS

Para este trabajo se ha elegido la plataforma xDEVS, desarrollada en el departamento de Arquitectura de Computadores y Automática de la Universidad Complutense de Madrid.

3.3. La plataforma xDEVS

XDEVS es una plataforma de modelado y simulación DEVS implementada en Java, cuyo núcleo está disponible en https://github.com/jlrisco/xdevs. En su implementación se cubre la creación de modelos atómicos y acoplados, extendiendo a las clases respectivas, y cuenta con las clases necesarias para poder realizar una simulación completa. La simulación se realiza por medio de Coordinadores que pueden ejecutar en tiempo real o virtual.

Para ilustrar el uso de xDEVS se muestran a continuación los ejemplos previos del juego del Ping-Pong: el modelo atómico del jugador `Player.java` y el modelo acoplado del juego `PingPong.java`.

Player.java

```
1  public class Player extends Atomic{
2
3      protected Port<Integer> iReceive = new Port<>("Receiver");
4      protected Port<Integer> oSend = new Port<>("Sender");
5      private double delay;
6
7      public Player(String name, double delay){
8          super(name);
9          this.delay = delay;
10         super.addInPort(iReceive);
11         super.addOutPort(oSend);
12     }
13
14     public void initialize() {
15         if(super.getName().equals("A")){
16             super.holdIn("Send", 0);
17         }else{
18             super.passivateIn("Wait");
19         }
20     }
21
22     public void exit() {
23     }
24
25     public void deltint() {
26         if(phaseIs("Send")){
27             super.holdIn("Wait", delay);
28         }else{
29             super.passivate();
30         }
31     }
32
33     public void deltext(double d) {
34         if(!phaseIs("Send")){
35             if(!iReceive.isEmpty() && iReceive.getSingleValue() != null){
36                 System.out.println("Player " + super.getName() + " received the
                       ball");
37                 super.holdIn("Send", delay);
```

```
            }else{
                super.passivate();
            }
        }
    }

    public void lambda() {
        if(phaseIs("Send")){
            System.out.println("Player " + super.getName() + " sent the ball");
            oSend.addValue(1);
        }else{
            oSend.addValue(null);
        }
    }
}
```

PingPong.java

```
public class PingPong extends Coupled{
    public PingPong(String name){
        super(name);
        Player pA = new Player("A", 1e-9);
        Player pB = new Player("B", 1e-9);

        super.addComponent(pA);
        super.addComponent(pB);

        super.addCoupling(pA.oSend, pB.iReceive);
        super.addCoupling(pB.oSend, pA.iReceive);
    }

    public static void main(String[] args) {
        PingPong game = new PingPong("Game");
        Coordinator coor = new Coordinator(game);
        coor.initialize();
        coor.simulate(100);
        coor.exit();
    }
}
```

Tras la compilación del modelo, la ejecución de la simulación nos devuelve la siguiente salida:

```
Player A sent the ball
Player B received the ball
Player B sent the ball
Player A received the ball
Player A sent the ball
Player B received the ball
Player B sent the ball
Player A received the ball
Player A sent the ball
Player B received the ball
...
```

Capítulo 4

Co-simulación con xDEVS y circuitos externos

Partiendo de los conocimientos necesarios de modelado y simulación en xDEVS, se pretende extender su funcionalidad incluyendo el uso de hardware externo. Tras un análisis de componentes necesarios y diversas pruebas, se llegó a la conclusión de que la co-simulación está determinada por:

- Un **Circuito externo** que cumpla con la función o funciones requeridas.
- Un protocolo de **comunicación** con el circuito externo.
- Un **driver**, o módulo del kernel, que gestione la comunicación.
- Uno o varios **modelos de XDEVS** que hagan de enlace entre el driver y el simulador.

Los códigos que se muestran en este trabajo son versiones simplificadas de las originales. Todos los códigos fuente de este proyecto se pueden encontrar en https://github.com/jlrisco/xdevs-lib/tree/master/src/xdevs/lib/tfgs/c1516/hwsw.

4.1. Circuito externo

El circuito externo en la simulación tendrá un papel funcional, es decir, cumplirá una función concreta necesaria para que la simulación funcione correctamente. Para el simulador xDEVS el trato con el circuito externo no será especial, sino que se le tratará como un modelo atómico más, dejando de esta manera el formalismo DEVS intacto.

En función de la simulación o la experimentación que se quiera realizar, el hardware externo puede variar. En este proyecto se utilizan circuitos lógicos externos montados sobre *breadboards* como componente hardware, pero pueden utilizarse FPGAs, procesadores de señal, etc. según la funcionalidad que se quiera conseguir y los medios disponibles.

Con el fin de facilitar la interconexión del circuito externo con el resto de las partes, se deben especificar bien las entradas y salidas que lo componen tal y como se muestra en la Figura 4.1.

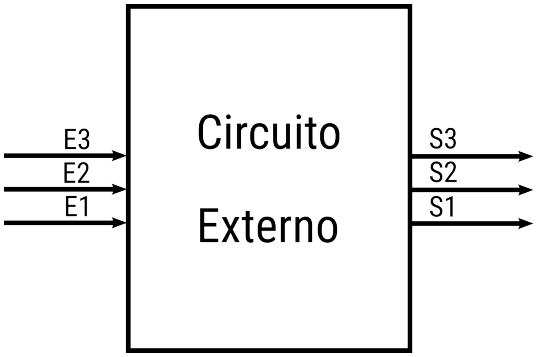

Figura 4.1: Abstracción de las entradas y salidas de un circuito externo.

4.2. Comunicación

Se debe especificar cómo se comunica el simulador con el circuito externo. Para ello se debe determinar:

- Comunicación **síncrona** o **asíncrona**.
- Envío de datos en **serie** o en **paralelo**.
- Protocolo de comunicación.
- Conexión entre las entradas y salidas del circuito externo y la RaspBerry.

Al ser un proyecto incremental, como ya se ha mencionado previamente, el objetivo no es realizar una comunicación fiable, robusta y/o segura, sino una comunicación sencilla para su posterior optimización en proyectos futuros. Por ello, en este proyecto, para comunicación entre la Raspberry Pi 2 y el circuito externo se utilizan los pines disponibles del periférico GPIO bajo una comunicación asíncrona en paralelo.

4.3. Driver

El driver se encarga de realizar la comunicación, previamente especificada, entre el simulador y el circuito externo en su totalidad.

La creación del driver se hace al estilo Linux, implementando un módulo del kernel en lenguaje *C* que recibe las órdenes del espacio de usuario, por medio de un fichero de dispositivo, y las conduce codificadas correctamente al circuito físico utilizando USB, GPIO, UART, etc. A continuación se muestra el esqueleto del código en C.

```
1 /* Reads the values from the user space and manages the communication with the
      extern circuit */
2 static ssize_t name_write(struct file *filp, const char __user *buf, size_t len
      , loff_t *off) {
3     char kbuff[BUF_LEN];
4     int val1, val2, ...;
5     /* Checks the buffer len */
6     if (len > BUF_LEN - 1) {
7         return -ENOSPC;
```

```
8       }
9       /* Transfer data from user to kernel space */
10      if (copy_from_user( &kbuff[0], buf, len ))
11          return -EFAULT;
12      kbuff[len] = '\0';
13      *off+=len;
14
15      /* Parse the buffer */
16      if(sscanf(&kbuff[0], "command %i %i ...", &val1, &val2, ...)){
17          /* Send to HW */
18          // ...
19      } else {
20          return -EINVAL;
21      }
22      return len;
23  }
24
25  /* Returns the hardware values to the user space */
26  static ssize_t name_read(struct file *filp, char __user *buf, size_t len,
        loff_t *off) {
27      int nr_bytes;
28      char kbuff[BUF_LEN];
29      kbuff[0] = '\0';
30
31      /* Reads HW */
32      // ...
33
34      /* Send data to the user space */
35      if (copy_to_user(buf, kbuff, nr_bytes))
36          return -EINVAL;
37
38      (*off)+=nr_bytes;  /* Update the file pointer */
39
40      return nr_bytes;
41  }
42
43  /* An application opens the device file */
44  static int name_open (struct inode *nodo, struct file *fich){
45      try_module_get(THIS_MODULE);
46      return SUCCESS;
47  }
48
49  /* An application closes the device file */
50  static int name_release (struct inode *nodo, struct file *fich){
51      module_put(THIS_MODULE);
52      return SUCCESS;
53  }
54
55  /* Proc operations implemented */
56  static const struct file_operations proc_entry_fops = {
57      .open = name_open,
58      .release = name_release,
59      .read = name_read,
60      .write = name_write,
61  };
62
63  /* Initializes the module */
64  int init_name_module( void )
65  {
66      /* Create the proc entry */
67      proc_entry_name = proc_create(module_name, 0666, NULL, &proc_entry_fops);
```

```
68      if (proc_entry_name == NULL) {
69          return -ENOMEM;
70      }
71      /* Init components */
72      // ...
73      return SUCCESS;
74  }
75
76  /* Finalizes the module */
77  void exit_name_module( void )
78  {
79      /* Remove proc entry */
80      remove_proc_entry(module_name, NULL);
81
82  }
83
84  module_init( init_name_module );
85  module_exit( exit_name_module );
```

Para poder utilizar los pines del periférico GPIO se hará uso de la librería del kernel `gpio.h`. A través de las funciones que proporciona se puede reservar pines, darles dirección de entrada o salida, escribir o leer el valor de un pin y liberarlos.

4.4. Modelos de xDEVS

Es necesario implementar al menos un modelo atómico de xDEVS que utilice el driver, a traves de un fichero de dispositivo, para integrar al circuito externo dentro de la simulación. De esta forma se encapsula al hardware, haciendo que el simulador lo trate como un modelo atómico más. A este modelo lo llamaremos M_D.

El envío de datos al fichero de dispositivo se realizará en las funciones δ_{int} y δ_{ext} de M_D, ya que la escritura del driver depende directamente del estado en el que se encuentre o del cambio en las entradas. La lectura del fichero de dispositivo se podrá realizar en las funciones λ, si la lectura del circuito debe ir directamente a una salida, o δ_{int}, si se requiere conocer datos del circuito externo en un estado concreto.

La especificación de los modelos debe seguir el formalismo DEVS, descrito previamente en el Capítulo 3 de este documento, de forma que posteriormente pueda implementarse en la plataforma preferida (xDEVS, PowerDEVS, etc).

Para incluir las interacciones con el hardware, en la especificación de los modelos, se utiliza F_D para representar al fichero de dispositivo junto al resto de componentes del modelo. La comunicación se representa de la siguiente forma:

$$F_D \leftarrow \{value_1, value_2, ...\}$$

$$\{value_1, value_2, ...\} \leftarrow F_D$$

En la primera fórmula se envían al fichero de dispositivo los valores del conjunto $\{value_1, value_2, ...\}$, y en la segunda se leen desde el mismo fichero de dispositivo el conjunto de valores. Por ejemplo, si en la función δ_{ext} se envían al circuito externo los valores V_1 y V_2, la especificación en el formalismo DEVS sería:

$$\delta_{ext}(phase, \sigma, \{V_1, V_2\}, t_e, \{(in1, V_1'), (in2, V_2')\}) = (nextphase, t_{FD}, F_D \leftarrow \{V_1', V_2'\})$$

4.5. Ejemplo práctico de co-simulación HW/SW

Como ejemplo de co-simulación, se va a modelar y simular un sistema Sumador sencillo. La simulación generará unas sumas que se tendrán que calcular en el circuito externo y se sacarán los resultados por la salida estándar.

4.5.1. Circuito externo

La parte que compone el hardware está compuesta únicamente por el circuito integrado *74LS283*, el cual realiza una suma entre dos operandos de 4 bits. La lógica interna se puede ver en la Figura 4.3 y la posición de los pines física en la Figura 4.2.

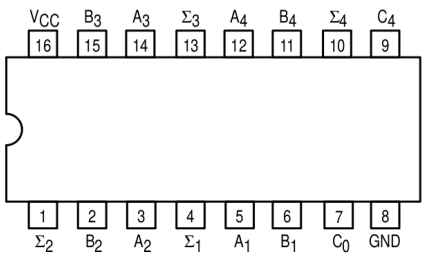

Figura 4.2: Pines del circuito integrado 74LS283.

Los pines $A_4A_3A_2A_1$ conforman el primer operando en binario, los pines $B_4B_3B_2B_1$ el segundo operando en binario, y los pines $\Sigma_4\Sigma_3\Sigma_2\Sigma_1$ conforman el resultado de la suma en binario. También se dispone de los pines C_0 y C_4 que conforman respectivamente el *Carry-in* y el *Carry-out* de la suma, de los cuales el primero estará conectado a tierra permanentemente y el segundo se utilizará como el bit más significativo de la suma. En la Figura 4.4 podemos ver la abstracción de las entradas y salidas del circuito.

4.5.2. Comunicación

Tratando de hacer un ejemplo lo más sencillo posible, se ha elegido una comunicación en paralelo asíncrona. Teniendo en cuenta que el retardo máximo que puede haber en el cálculo del circuito integrado 74LS283 es de 24ns [Cuadro 4.1], para la comunicación se seguirá el siguiente protocolo:

1 Envío de operandos A y B.

2 Espera de $1ms$ [1].

[1] Tiempo suficiente para asegurar la resolución del SO

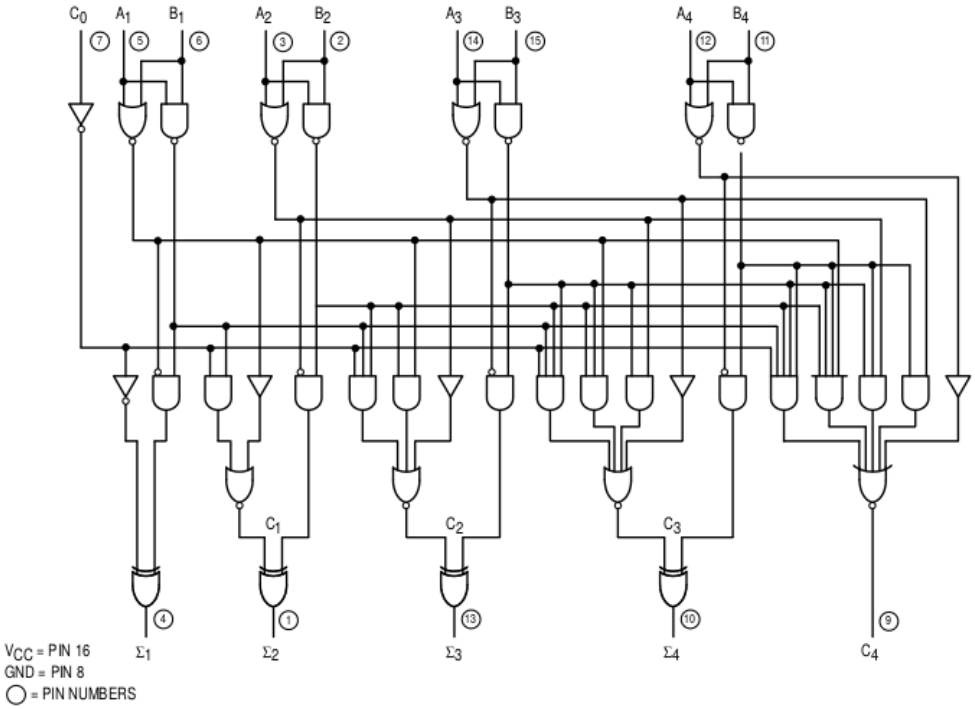

Figura 4.3: Circuito lógico del IC 74LS283.

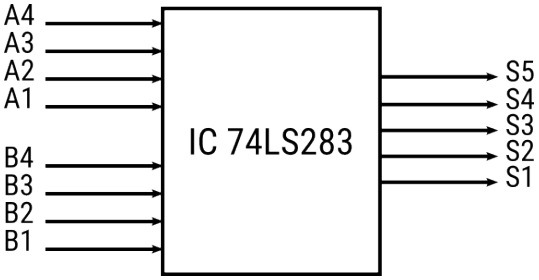

Figura 4.4: Abstracción de entradas y salidas.

3 Lectura del resultado Σ.

Los pines del módulo GPIO de la Raspberry Pi utilizados para el ejemplo se incluyen en el Cuadro 4.2 y se pueden ver interconectados en la Figura 4.5 con las entradas y salidas del circuito.

4.5.3. Driver

El driver contiene principalmente las funciones de escritura, para enviar los operandos de la suma, y de lectura, para leer el resultado.

Los pasos que se siguen en la operación de **escritura** son los siguientes:

1 - Reconoce el comando *add* A B.

Symbol	Limits			Unit
	Min	Typ	Max	
$t_{PLH}\ C_0 \rightarrow \Sigma$	-	16	24	ns
$t_{PHL}\ C_0 \rightarrow \Sigma$	-	15	24	ns
$t_{PLH}\ A, B \rightarrow \Sigma$	-	15	24	ns
$t_{PHL}\ A, B \rightarrow \Sigma$	-	15	24	ns
$t_{PLH}\ C_0 \rightarrow C_4$	-	11	17	ns
$t_{PHL}\ C_0 \rightarrow C_4$	-	11	22	ns
$t_{PLH}\ A, B \rightarrow C_4$	-	11	17	ns
$t_{PHL}\ A, B \rightarrow C_4$	-	12	17	ns

Cuadro 4.1: Tabla de retardos del IC 74LS283

SIGNAL	GPIO Pin	74LS283 Pin
A_4	06	12
A_3	13	14
A_2	19	03
A_1	26	05
B_4	14	11
B_3	22	15
B_2	27	02
B_1	17	06
S_5	24	09
S_4	23	10
S_3	18	13
S_2	15	01
S_1	05	04

Cuadro 4.2: Conexiones entre GPIO y IC 74LS283 para las distintas señales.

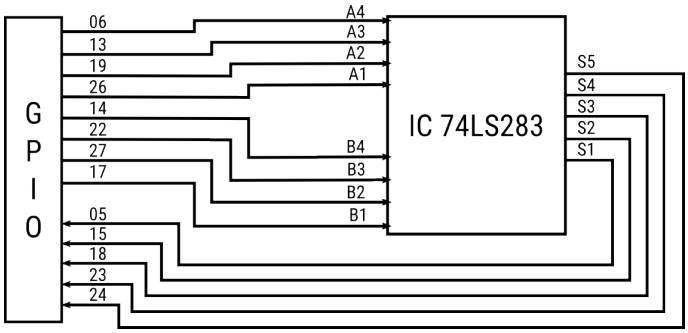

Figura 4.5: Conexión entre circuito y GPIO.

2 - Convierte los valores A y B a binario.

3 - Envía los operandos por los pines de la GPIO para realizar la suma.

4 - Espera $1\mu s$.

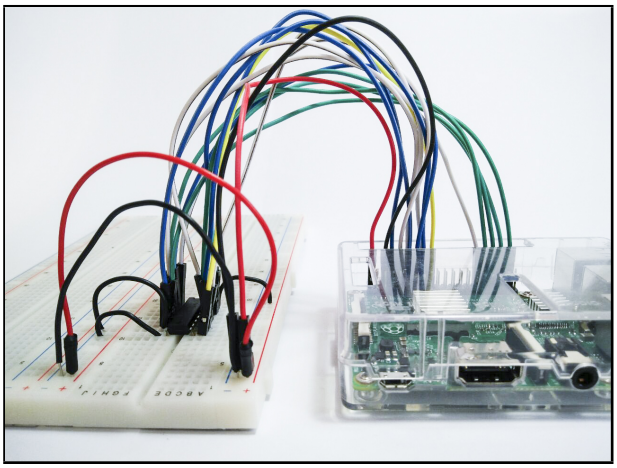

Figura 4.6: Raspberry Pi 2 B conectada a IC 74LS283.

5 - Lee por los pines de la GPIO el resultado de la suma y lo guarda.

Operación de escritura

```
static ssize_t sumador4bit_write(struct file *filp, const char __user *buf,
    size_t len, loff_t *off) {
    char kbuff[BUF_LEN];          //Userspace buffer copy
    int numA, numB;
    int A[4] = {0,0,0,0}, B[4] = {0,0,0,0};
    {...}
    /* Transfer data from user to kernel space */
    if (copy_from_user( &kbuff[0], buf, len ))
        return -EFAULT;
    {...}
    /* Sets the sum to processing */
    sum = S4B_PROCESSING;
    /* Parse the buffer */
    if(sscanf(&kbuff[0],"add %i %i\n", &numA, &numB)){
        {...}
        /* Send A and B */
        decimalToArray(numA, A);
        decimalToArray(numB, B);
        sendValues(A, B);
        /* Wait for the sum and read C*/
        udelay(1);
        sum = getC();
        {...}
    } else {    /* ERROR */
        {...}
    }
    return len;
}
```

En la operación de **lectura** únicamente se devuelve el valor de la suma, leído previamente en la operación de escritura.

Operación de lectura

```
static ssize_t sumador4bit_read(struct file *filp, char __user *buf, size_t len
    , loff_t *off) {
    int nr_bytes;
    char kbuff[BUF_LEN];
    {...}
    /* Charges the C value */
    sprintf(kbuff, "%i", sum);
    nr_bytes = strlen(kbuff);
    /* Send data to the user space */
    if (copy_to_user(buf, kbuff, nr_bytes))
        return -EINVAL;
    {...}
    return nr_bytes;
}
```

Para generar el driver y poder utilizarlo hay que compilarlo e introducirlo en el kernel. La compilación del módulo se hará utilizando el siguiente `Makefile`.

```
obj-m +=  sumador4bit.o
all:
     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
     make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Desde la carpeta que contiene el fichero `Makefile` y el código del módulo implementado, `sumador4bit.c`, se ejecuta el comando `make` para hacer la compilación. Se generará entonces el objeto `sumador4bit.ko`, que será el fichero del módulo insertable en el kernel a través del comando `insmod`.

Finalmente, se realizan una serie de pruebas para confirmar que cumple con su función correctamente con los comandos `echo` y `cat`, como en el ejemplo siguiente:

```
$ echo add 4 10 > /proc/sumador4bit
$ cat /proc/sumador4bit
14
```

4.5.4. Modelos de xDEVS

En el simulador se necesita un modelo atómico *Adder* que haga de enlace hacia el driver, otro modelo atómico *AdderMatrix* que le pase operandos al primero para que realice la suma y un modelo acoplado *AddTest* que contenga a ambos e interconecte sus entradas y salidas, tal y como se muestra en la Figura 4.7.

Modelo Adder

Descripción según el formalismo DEVS del modelo atómico *Adder*:

$$Adder = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

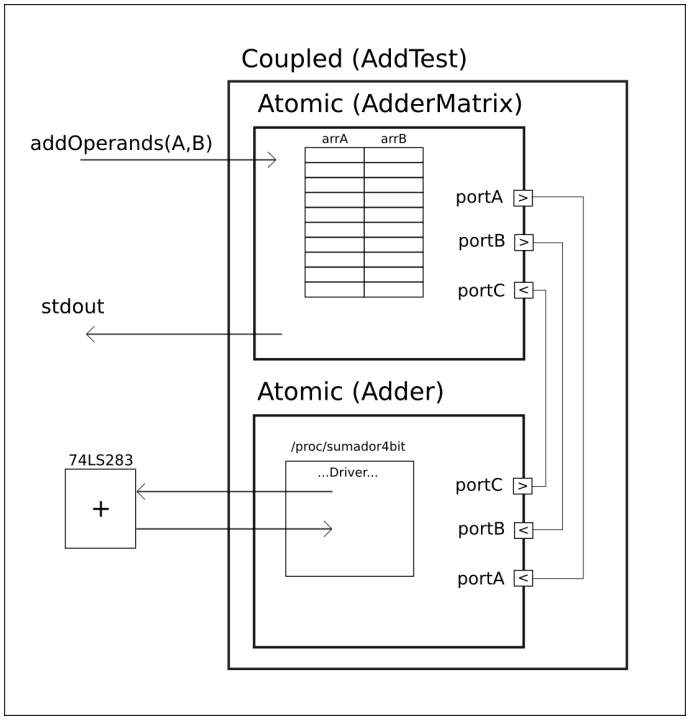

Figura 4.7: Representación gráfica del modelado DEVS

- $X = \{(iA, A), (iB, B)\}$, con $A, B \in \mathbb{N}_0^{15}$
- $S = \{(s, \sigma, \{A, B\}) \mid s \in \{"passive", "read", "active"\}, \sigma \in \mathbb{R}_0^+\}$
- $Y = \{(oC, C)\}$, con $C \in \mathbb{N}_0^{30}$
- $\delta_{int}("active", \sigma, \{A, B\}) = ("passive", \infty, \emptyset)$
  $\delta_{int}("read", \sigma, \{A, B\}) = ("active", \sigma, \emptyset) \Rightarrow (\{C\} \leftarrow F_D)$
- $\delta_{ext}(phase, \sigma, \{A, B\}, t_e, \{(iA, A'), (iB, B')\}) = ("read", \sigma, F_D \leftarrow \{A', B'\})$
- $\lambda("active", \sigma, \{A, B\}) = \{C\}$

Tal y como se ha especificado, el modelo atómico *Adder* tiene dos puertos de entrada A y B, correspondientes a los operandos de la suma, y un puerto de salida C, correspondiente al resultado. Constará de 3 estados:

- $Passive$: El modelo está a la espera de cambios en A y B. Al suceder un evento envía el comando *Add A B* al driver.
- $Read$: El modelo lee del fichero de dispositivo el resultado de la suma.
- $Active$: El modelo tiene un resultado calculado. Se duerme a la espera de cambios en las entradas.

Ahora se muestra la implementación de las funciones principales del modelo en xDEVS. En un primer momento, el modelo *Adder* permanece dormido hasta recibir nuevos operandos.

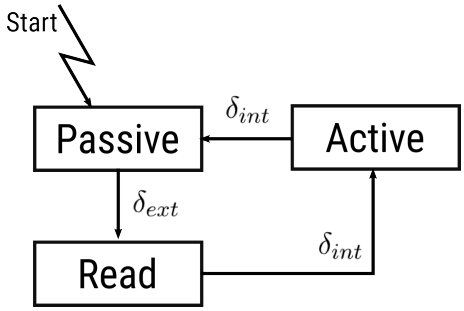

Figura 4.8: Diagrama de estados del modelo atómico *Adder*.

```
1       public void initialize() {
2           super.passivate();
3       }
```

Los eventos de entrada, o cambios en A y B, hacen que se ejecute la función δ_{ext} y se escriba en el fichero de dispositivo del driver el comando *add* A B. Seguidamente se cambia al estado *read*.

```
1       public void deltext(double d) {
2           if(inputHasChanged()){
3               readValues(valA, valB);
4               String cmd = "add " + this.valA + " " + this.valB + "\n";
5               driver.write(cmd);
6               super.holdIn("read", delay);
7           }
8       }
```

La función δ_{int}, en caso de que el estado sea *read*, lee el resultado de la suma, que será proporcionado por el driver al leer del fichero de dispositivo. Si el estado es *active*, se dormirá al modelo a la espera de nuevos operandos.

```
1       public void deltint() {
2           if(phaseIs("active")){
3               valC = null;
4               super.passivate();
5           }else if(phaseIs("read")){
6               this.valA = null;
7               this.valB = null;
8               String result = driver.read();
9               valC = Integer.parseInt(result);
10              super.holdIn("active", delay);
11          }
12      }
```

La función λ cambia la salida C con el resultado leído en cada momento.

```
1       public void lambda() {
2           portC.addValue(valC);
3       }
```

Modelo AdderMatrix

Descripción según el formalismo DEVS del modelo atómico *AdderMatrix*:

$$AdderMatrix = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

- $X = \{(iC, C)\}$, con $C \in \mathbb{N}_0^{30}$
- $S = \{(s, \sigma, C) \mid s \in \{"active", "send", "wait", "read"\}, \sigma \in \mathbb{R}_0^+\}$
- $Y = \{(oA, A), (oB, B)\}$, con $A, B \in \mathbb{N}_0^{15}$
- $\delta_{int}("active", \sigma, C) = ("send", \sigma, \emptyset)$ si $(M \neq \emptyset)$, con M $\equiv$ Matriz de operandos.
  $\delta_{int}("active", \sigma, C) = ("passive", \infty, \emptyset)$, si $(M = \emptyset)$
  $\delta_{int}("send", \sigma, C) = ("wait", \sigma, \emptyset)$
  $\delta_{int}("wait", \sigma, C) = ("passive", \infty, \emptyset)$
  $\delta_{int}("read", \sigma, C) = ("active", \sigma, \emptyset)$
- $\delta_{ext}(phase, \sigma, C, t_e, C') = ("read", \sigma, C')$
- $\lambda(phase, \sigma, C) = \{(M_{i0}, M_{i1}) \mid i \in \mathbb{N}_0^{10}\}$

El modelo atómico, *AdderMatrix*, contiene una matriz de operandos y va enviando A y B como eventos de salida de dos en dos. En contrapartida al modelo anterior, este modelo recibirá el resultado de la suma C como entrada. Tendrá los estados:

- $active$: Si quedan operandos por enviar al sumador, pasa al estado $send$; sino, se duerme.
- $send$: Carga los siguientes valores de A y B desde la matriz de operandos para ser enviados por las salidas respectivas.
- $wait$: Duerme al modelo para ser despertado por el sumador al recibir el resultado de la suma.
- $read$: En este estado se ha leído el resultado, se actualiza el índice de la matriz de operandos y se pasa de nuevo al estado $active$.

A diferencia del modelo *Adder*, este modelo debe iniciar la comunicación enviando los operandos A y B. Las funciones δ_{int} y δ_{ext} realizan la transición entre estados del modelo hasta haber enviado todos los operandos por medio de la función λ. A continuación se muestra la implementación en xDEVS de las funciones principales.

```
1      public void deltint() {
2          if(phaseIs("active")){
3              if(current < arrA.size()){
4                  super.holdIn("send", delay);
5              }else{
6                  super.passivate();
7              }
8          }else if(phaseIs("send")){
9              valA = arrA.get(current);
10             valB = arrB.get(current);
11             System.out.println("Sent " + valA + " + " + valB);
```

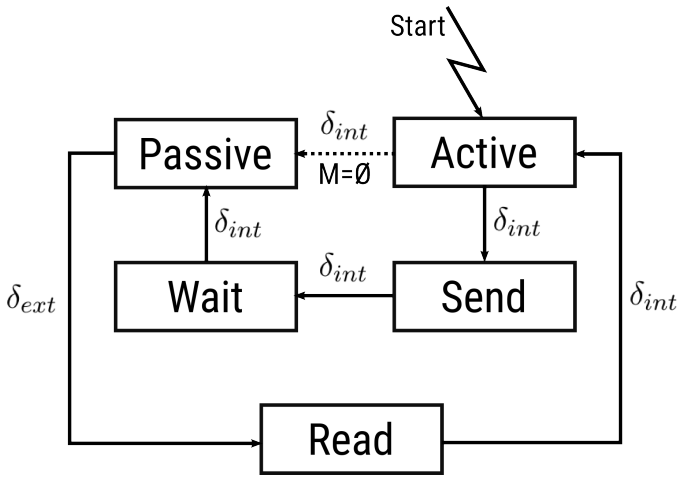

Figura 4.9: Diagrama de estados del modelo atómico *AdderMatrix*.

```
12              super.holdIn("wait", delay);
13          }else if(phaseIs("wait")){
14              valA = null;
15              valB = null;
16              super.passivate();
17          }else if(phaseIs("read")){
18              current++;
19              super.holdIn("active", delay);
20          }
21      }
22      public void deltext(double d) {
23          if(!portC.isEmpty() && portC.getSingleValue() != null){
24              valC = portC.getSingleValue();
25              System.out.println("Received " + valC);
26              super.holdIn("read", delay);
27          }
28      }
29      public void lambda() {
30          portA.addValue(valA);
31          portB.addValue(valB);
32      }
```

Modelo AddTest

Descripción según el formalismo DEVS del modelo acoplado *AddTest*:

$$AddTest = \langle X, Y, C_i, EIC, IC, EOC \rangle$$

- $X = \emptyset$
- $Y = \emptyset$
- $C_{Adder} : \{adder\}$
  $C_{AdderMatrix} : \{matrix\}$
- $EIC = \emptyset$

- $IC = \{(adder.oC, matrix.iC), (matrix.oA, adder.iA), (matrix.oB, adder.iB)\}$
- $EOC = \emptyset$

El modelo acoplado, *AddTest*, contiene a los dos modelos atómicos, *Adder* y *AdderMatrix*, y realiza la conexión entre sus puertos. Va a ser el encargado de coordinarlos en la simulación. En xDEVS se implementa como sigue.

```
1     public AddTest(String name){
2         super(name);
3         adder = new Adder("adder");
4         addComponent(adder);
5         matx = new AdderMatrix("matrix");
6         addComponent(matx);
7
8         addCoupling(matx.portA, adder.portA);
9         addCoupling(matx.portB, adder.portB);
10        addCoupling(adder.portC, matx.portC);
11    }
```

El método principal, o *main*, de la simulación debe inicializar la matriz de operandos con números aleatorios e iniciar la simulación con un coordinador (`Coordinator`) que contenga al modelo acoplado *AddTest*.

```
1     public static void main(String[] args) {
2         AddTest test = new AddTest("test");
3         Random rdm = new Random();
4         for(int i = 0; i < 10; i++)
5            test.addOperands(Math.abs(rdm.nextInt() % 15), Math.abs(rdm.nextInt
                 () % 15));
6         Coordinator coor = new Coordinator(test);
7         coor.initialize();
8         coor.simulate(Long.MAX_VALUE);
9         coor.exit();
10    }
```

4.5.5. Resultados de la simulación

Al compilar los modelos xDEVS se genera el archivo `rpi-adder.jar`. La simulación se lanza utilizando el archivo a través del comando:

```
$ java -jar rpi-adder.jar
```

Y, gracias a la implementación de *AdderMatrix*, la salida por pantalla nos muestra los resultados de la simulación:

```
Sent 2 + 11
Received 13
Sent 13 + 5
Received 18
Sent 2 + 12
Received 14
Sent 14 + 4
```

```
Received 18
Sent 4 + 11
Received 15
Sent 11 + 3
Received 14
Sent 5 + 1
Received 6
Sent 11 + 6
Received 17
Sent 9 + 2
Received 11
Sent 5 + 4
Received 9
```

4.6. Conclusiones

Para poder realizar co-simulaciones entre hardware y software, a través de la plataforma xDEVS, se necesita interactuar con un fichero de dispositivo. La lectura y escritura en el fichero de dispositivo se realiza con órdenes internas de *Java*, por lo que el simulador no va a notar diferencia alguna entre el modelo atómico que trata al hardware (M_D) y los demás. Los comandos enviados al fichero de dispositivo serán clasificados por el driver, y enviará la información necesaria al circuito externo por medio del GPIO. La lectura del fichero de dispositivo hará que el driver lea el valor devuelto por el circuito externo y lo retornará al modelo del simulador M_D.

Este proceso de co-simulación consigue que no se deba modificar el formalismo DEVS, de forma que seguirá cumpliendo todas las propiedades matemáticas que ya cumplía.

En el ejemplo del sumador, la ejecución en tiempo real no es necesaria, ya que es el driver quien realiza la espera al circuito externo, obligando al simulador a esperar. Sin embargo, este ejemplo sólo se realiza para dar a entender el funcionamiento de la co-simulación; los retardos se deben especificar en los modelos DEVS, obligando a la ejecución a ser en tiempo real, tal y como se muestra en el Capítulo 5.

Capítulo 5

Co-simulación del circuito controlador de un ascensor

Como caso de estudio de este trabajo, se va a proceder a simular un ascensor con circuitos integrados como hardware externo. Los códigos del driver y de los modelos xDEVS mostrados en este capítulo son versiones simplificadas de los originales. Todo el código fuente original de este trabajo se puede consultar en https://github.com/jlrisco/xdevs-lib/tree/master/src/xdevs/lib/tfgs/c1516/hwsw.

La simulación completa en xDEVS del circuito del ascensor será el punto de partida. A continuación se irán sacando, uno a uno, los circuitos integrados del simulador para que formen parte del hardware externo, de forma que se van a desarrollar cuatro co-simulaciones diferentes en total. Al final, en la cuarta co-simulación, se tendrá el circuito completo del ascensor como hardware externo y el simulador gestionará los estímulos que se le envían, enviará la señal de reloj y sacará por pantalla la planta actual del ascensor.

Esta simulación debe ser en tiempo real y los retardos de los circuitos integrados serán definidos en los modelos atómicos que los gestionen. Para ello, a diferencia del ejemplo del Sumador, se utiliza un coordinador de tipo `RTCentralCoordinator`.

Antes de comenzar a elaborar las co-simulaciones entre hardware y software, se debe tener finalizado:

- El diseño del circuito lógico usado en este capítulo: el controlador de un ascensor de siete plantas.
- La simulación completa en xDEVS del ascensor.

En base al diseño lógico y a la simulación del ascensor, las partes que van a componer el proceso son las siguientes:

1 - Co-simulación de un ascensor con xDEVS y el circuito integrado *74LS283* (Sumador).

2 - Co-simulación de un ascensor con xDEVS y los circuitos integrados *74LS283* (Sumador) y *74LS04* (NOT).

3 - Co-simulación de un ascensor con xDEVS y los circuitos integrados *74LS283* (Sumador), *74LS04* (NOT) y *74LS10* (NAND).

4 - Co-simulación de un ascensor con xDEVS y los circuitos integrados *74LS283* (Sumador), *74LS04* (NOT), *74LS10* (NAND) y *74LS169* (Contador).

5.1. Circuito lógico del controlador

Se parte de una de las prácticas de la asignatura Fundamentos de Computadores, de la Facultad de Informática de la Universidad Complutense de Madrid, que consta del diseño y montaje de un circuito lógico secuencial síncrono que se comporte como el ascensor de una vivienda de siete plantas.

En la Figura 5.1 se puede ver el diseño final del circuito lógico. Para el diseño se debe utilizar una parte de los circuitos integrados disponibles en el laboratorio, en concreto los circuitos integrados *74LS04* (Puertas NOT), *74LS10* (Puertas NAND), *74LS283* (Sumador 4 bits) y *74LS169* (Contador 4 bits). El contador se encargará de incrementar o decrementar el piso actual. El sumador y las puertas NOT y NAND, se encargarán de la comparación entre el piso actual y el solicitado, y generarán las señales necesarias para que el contador dirija la cuenta hacia el piso adecuado.

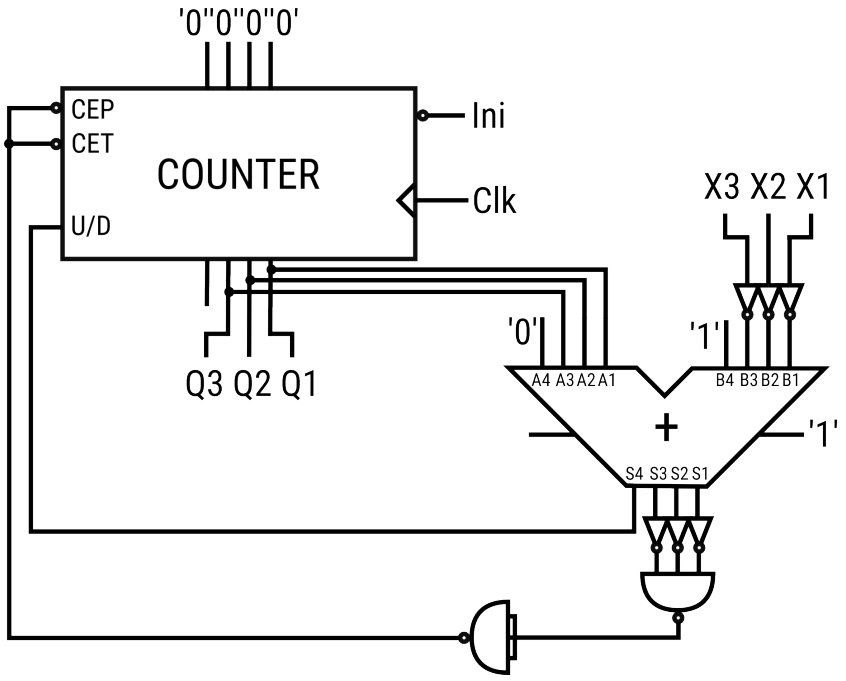

Figura 5.1: Circuito lógico secuencial de un ascensor de siete plantas.

Las entradas del circuito son:

- $X_3X_2X_1$: Contienen el piso destino en binario.
- Clk : Reloj del sistema.
- Ini : Señal que reinicia el contador. Pone al ascensor en el piso 0.

Las salidas son:

- $Q_3Q_2Q_1$: Indican el piso en el que se está actualmente en binario.

Para detectar si se ha llegado al piso destino, se hace una resta entre el piso actual y el piso deseado, si el resultado es 0 se habrá llegado al destino. Para hacer la resta se transforma la entrada $X_3X_2X_1$ a complemento-2 para ser introducido así en el sumador. Las salidas del sumador, $S_4S_3S_2S_1$, son utilizadas para generar las señales U/D, CEP y CET que gestionan la cuenta del contador.

En cada parte del proceso de la co-simulación del ascensor se verá en detalle el circuito externo y las conexiones entre los pines del mismo.

5.2. Simulación de un ascensor con xDEVS

Para realizar la co-simulación entre hardware y software, se parte de la simulación completa en tiempo real[1] del circuito lógico del ascensor en xDEVS. Para ello se debe tener un modelo atómico por cada circuito integrado que compone el diseño, tratando los pines como puertos de entrada o salida según les corresponda. Cabe destacar que los modelos serán implementados como sistemas de estado continuo, es decir, cada modelo atómico estará siempre a la espera de un cambio en las entradas para actuar y cambiar sus salidas. Por ello la función que contiene toda la actividad interna es δ_{ext}. Los modelos atómicos, junto a sus puertos de entrada y salida, son:

- $StimulusFile$: Fichero de estímulos.
  - $sw7$: Entrada de reinicio del sistema.
  - $sw2, sw1, sw0$: Entrada del piso destino.
- $Clock$: Reloj del sistema.
  - $oClk$: Salida del reloj.
- $Constant$: Valor constante para Vcc y Gnd.
  - out : Salida del valor constante.
- $IC74169$: Circuito integrado del contador.
  - $pin10, pin7$: (CET,CEP) Entrada para habilitar la cuenta (1) o pararla (0).
  - $pin1$: (U/D) Entrada para cuenta creciente (1) o decreciente (0).
  - $pin6, pin5, pin4, pin3$: (P3,P2,P1,P0) Entrada de valor para carga paralela.
  - $pin9$: (PE) Entrada para habilitar la carga paralela.
  - $pin2$: (CP) Entrada de reloj.
  - $pin16$: Entrada de Vcc.
  - $pin8$: Entrada de Gnd.
  - $pin11, pin12, pin13, pin14$: (Q3,Q2,Q1,Q0) Salida del valor actual del contador.
- $IC74283$: Circuito integrado del sumador.
  - $pin12, pin14, pin3, pin5$: (A4,A3,A2,A1) Entrada del operando A.
  - $pin11, pin15, pin2, pin6$: (B4,B3,B2,B1) Entrada del operando B.

[1] La co-simulación del ascensor debe ser en tiempo real. En tiempo virtual podría fallar, ya que los tiempos de retardo del circuito externo se especifican en los modelos xDEVS.

  - $pin7$: (C0) Entrada del *Carry-in*.
  - $pin9$: (C4) Salida del *Carry-out*.
  - $pin16$: Entrada de Vcc.
  - $pin8$: Entrada de Gnd.
  - $pin10, pin13, pin1, pin4$: (S4,S3,S2,S1) Salida del valor actual del sumador.
- $IC7410$: Circuito integrado de puertas NAND.
  - $pin13, pin1, pin2$: Entradas de NAND 1.
  - $pin12$: Salida de NAND 1.
  - $pin9, pin10, pin11$: Entradas de NAND 2.
  - $pin8$: Salida de NAND 2.
  - $pin3, pin4, pin5$: Entradas de NAND 3.
  - $pin6$: Salida de NAND 3.
  - $pin14$: Entrada de Vcc.
  - $pin7$: Entrada de Gnd.
- $IC7404$: Circuito integrado de puertas NOT.
  - $pin1, pin3, pin5, pin13, pin11, pin9$: Entradas de las puertas NOT.
  - $pin2, pin4, pin6, pin12, pin10, pin8$: Salidas respectivas de las puertas NOT.
  - $pin16$: Entrada de Vcc.
  - $pin8$: Entrada de Gnd.
- $Console$: Encargado de imprimir por pantalla un valor cuando se modifica.
  - in : Entrada del valor para imprimir por pantalla.

Tras disponer de los atómicos necesarios, alojados en https://github.com/jlrisco/xdevs-lib/tree/master/src/xdevs/lib/logic, se elabora un modelo acoplado, *Elevator*, que los contenga a todos y aplique las conexiones entre los puertos para cumplir con el diseño del circuito lógico, mostrado previamente en la Figura 5.1. Su descripción en el formalismo DEVS es la siguiente:

$$Elevator = \langle X, Y, C_i, EIC, IC, EOC \rangle$$

- $X = \emptyset$
- $Y = \emptyset$
- $C_{StimulusFile} = \{stimulus\}$
  $C_{Clock} : \{clock\}$
  $C_{Constant} : \{vcc, gnd\}$
  $C_{IC74169} : \{counter\}$
  $C_{IC74283} : \{adder\}$
  $C_{IC7410} : \{nands\}$
  $C_{IC7404} : \{nots\}$
  $C_{Console} : \{console2, console1, console0\}$

- $EIC = EOC = \emptyset$
- $IC = \{(nots.pin2, nands.pin3)$, $(nots.pin4, nands.pin4)$, $(nots.pin6, nands.pin5)$, $(nots.pin8, adder.pin6)$, $(nots.pin10, adder.pin2)$, $(nots.pin12, adder.pin15)$, $(nands.pin6, nands.pin1)$, $(nands.pin6, nands.pin2)$,$(nands.pin12, counter.pin7)$, $(nands.pin12, counter.pin10)$, $(adder.pin1, nots.pin3)$, $(adder.pin4, nots.pin5)$, $(adder.pin10, counter.pin1)$, $(adder.pin13, nots.pin1)$, $(counter.pin12, console2.in)$, $(counter.pin12, adder.pin14)$, $(counter.pin13, console1.in)$, $(counter.pin13, adder.pin3)$, $(counter.pin14, console0.in)$, $(counter.pin14, adder.pin5)$, $(vcc.out, counter.pin16)$, $(vcc.out, adder.pin7)$, $(vcc.out, adder.pin11)$, $(vcc.out, adder.pin16)$, $(vcc.out, nands.pin13)$, $(vcc.out, nands.pin14)$, $(vcc.out, nots.pin14)$, $(gnd.out, counter.pin3)$, $(gnd.out, counter.pin4)$, $(gnd.out, counter.pin5)$, $(gnd.out, counter.pin6)$, $(gnd.out, counter.pin8)$, $(gnd.out, adder.pin8)$, $(gnd.out, adder.pin12)$, $(gnd.out, nands.pin7)$, $(gnd.out, nots.pin7)$, $(stimulus.sw7, counter.pin9)$, $(stimulus.sw2, nots.pin13)$,$(stimulus.sw1, nots.pin11)$, $(stimulus.sw0, nots.pin9)$, $(stimulus.stop, clock.stop)\}$

Para la implementación del modelo acoplado en xDEVS, únicamente se deben añadir los modelos atómicos en el constructor del modelo acoplado y declarar las conexiones entre ellos con el método `addCoupling`.

Simulación y resultados

Gracias a los componentes de tipo *Console*, se irá sacando por pantalla la información que se le haya conectado. Es decir, en la simulación saldrá por pantalla el piso que proporcione el contador, en el momento en el que el piso varíe, con el formato Q_i*:tiempo:valor*.

Como fichero de estímulos para la simulación se utilizará `Test1.txt`. Su funcionalidad es hacer uso del ascensor, como si fuera una persona, de forma programada. El contenido del fichero se muestra a continuación, y su comportamiento es únicamente ir al cuarto piso en el instante 4 e ir al segundo piso en el instante 10.

Test1.txt

```
# This must be replaced by an XML file, in future versions
# BEGIN PORTS ------------------------------------ #
# Port names
sw7:java.lang.Integer
sw2:java.lang.Integer
sw1:java.lang.Integer
sw0:java.lang.Integer
stop:java.lang.Integer
# END PORTS -------------------------------------- #
# BEGIN VALUES ----------------------------------- #
# Initial values:
00:00:00.000;sw7;0
00:00:00.000;sw2;0
00:00:00.000;sw1;0
00:00:00.000;sw0;0
00:00:02.000;sw7;1
# Go to 4th floor
00:00:04.000;sw2;1
00:00:04.000;sw1;0
00:00:04.000;sw0;0
# Go to 2nd floor
```

```
00:00:10.000;sw2;0
00:00:10.000;sw1;1
00:00:10.000;sw0;0
# Stop the simulation
00:00:15.000;stop;1
# END VALUES ------------------------------------- #
```

Al realizar la simulación se aprecia que a partir del instante 4 el ascensor comienza a subir hasta alcanzar el piso destino. Posteriormente, a partir del instante 10 el ascensor comienza a descender hasta el nuevo piso destino. La salida por pantalla, modificada para una mejor visión, es la siguiente:

```
# t = 0
Q2::0.0:0
Q1::0.0:0
Q0::0.0:0

Q2::0.5:0
Q1::0.5:0
Q0::0.5:0

# t = 1
Q2::1.5:0
Q1::1.5:0
Q0::1.5:0

# t = 4
Q2::4.5:0
Q1::4.5:0
Q0::4.5:1

# t = 5
Q2::5.5:0
Q1::5.5:1
Q0::5.5:0

# t = 6
Q2::6.5:0
Q1::6.5:1
Q0::6.5:1

# t = 7
Q2::7.5:1
Q1::7.5:0
Q0::7.5:0

# t = 10
Q2::10.5:0
Q1::10.5:1
Q0::10.5:1

# t = 11
Q2::11.5:0
Q1::11.5:1
```

```
Q0::11.5:0
```

5.3. xDEVS - 74LS283

Partiendo de los modelos xDEVS implementados previamente, se va a extraer el circuito integrado *IC74283*, o sumador de 4 bits, del simulador para pasar a formar el hardware externo. En consecuencia, se debe crear un nuevo modelo atómico, `MDCosim1.java`, que hará de enlace entre el simulador y el driver del circuito externo, `rpi-cosim1.c`.

Circuito externo y comunicación

Como circuito externo se utiliza el *IC74LS283*, cuya lógica interna se ha mostrado previamente, en el Capítulo 4, en la Figura 4.3 y la posición de sus pines en la Figura 4.2. La función que debe cumplir es la de llevar a cabo la resta que sirve de comparación entre el piso destino y el piso actual. Las entradas y salidas que componen el circuito son:

- $A_3A_2A_1$: Operando que guarda el piso actual.
- $B_3B_2B_1$: Operando que recibe el piso destino en complemento-1.
- $S_3S_2S_1$: Resultado de la resta para utilizar como habilitador de la cuenta.
- S_4 : Bit más significativo del resultado de la resta. Indica si la cuenta será ascendente o descendente.

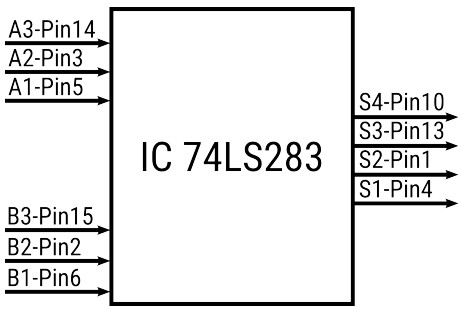

Figura 5.2: Abstracción de entradas y salidas del circuito externo.

Para cumplir con el diseño del ascensor, mostrado en la Figura 5.1, en el circuito externo se deben conectar las señales A_4 y C_{in} a Vcc y la señal B_4 a tierra o Gcc.

Siguiendo con la intención de hacer una comunicación sencilla, se realizará de forma paralela asíncrona. Los datos $A_3A_2A_1$ y $B_3B_2B_1$ serán enviados por el driver en su función de escritura y los datos $S_4S_3S_2S_1$ serán leídos en su función de lectura. Para esta comunicación, será el modelo atómico del simulador el encargado de esperar, al menos, 24ns para dar tiempo al circuito integrado a realizar la suma correctamente.

La conexión entre los pines del periférico GPIO y el circuito integrado 74LS283 se puede ver en el Cuadro 5.1 y el acabado final en la Figura 5.4 y Figura 5.5.

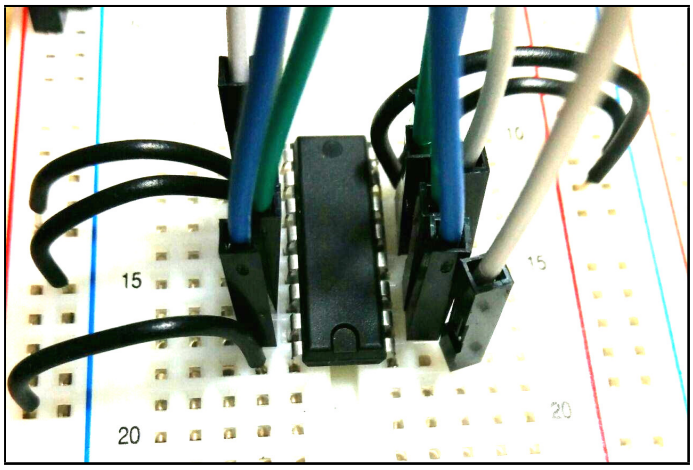

Figura 5.3: Circuito externo con conexiones finalizadas.

SIGNAL	GPIO Pin	74LS283 Pin
A_4	$3V3$	12
A_3	14	14
A_2	15	03
A_1	18	05
B_4	Gcc	11
B_3	25	15
B_2	08	02
B_1	07	06
S_4	12	10
S_3	16	13
S_2	20	01
S_1	21	04
C_{in}	$3V3$	07

Cuadro 5.1: Conexiones entre GPIO y IC 74LS283 para las distintas señales.

Driver

El driver constará principalmente de las operaciones de escritura y de lectura en el fichero de dispositivo. En la escritura se enviará al circuito externo, por los pines del GPIO, las señales recibidas del simulador. El formato del dato que puede recibir el driver en la escritura es un número en binario, de la forma *0bxxxxxx*, donde los 3 primeros bits corresponden al operando del piso destino, $B_3B_2B_1$, y los 3 últimos al operando del piso actual, $A_3A_2A_1$.

Función de escritura

```
1  static ssize_t elevator_write(struct file *filp, const char __user *buf, size_t
       len, loff_t *off) {
2      char kbuff[BUF_LEN];
3      int value, rest, temp, i, data;
4      {...}
5      /* Transfer data from user to kernel space */
6      if (copy_from_user( &kbuff[0], buf, len ))
```

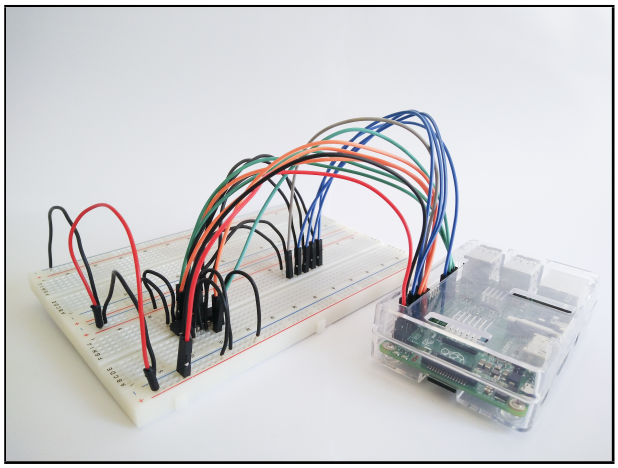

Figura 5.4: Raspberry Pi 2B y Circuito externo - 1

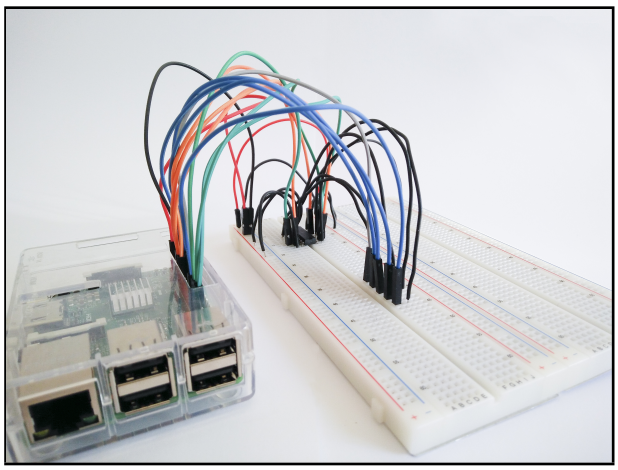

Figura 5.5: Raspberry Pi 2B y Circuito externo - 2

```
7           return -EFAULT;
8       {...}
9       /* Parse the buffer */
10      if(sscanf(&kbuff[0],"0b%d", &value)){
11          temp = value;
12          i=0;
13          /* Send the values to the GPIO */
14          while(i < 6){
15              rest = temp % 10;
16              data = (rest == 0) ? 0 : 1;
17              gpio_set_value(dataSPin[i], data);
18              temp /= 10;
19              i++;
20          }
21          printk(KERN_INFO "Elevator: Sent 0b%i\n", value);
22      } else {    /* ERROR */
23          {...}
24      }
25      return len;
26  }
```

En la lectura se leerá por los pines del GPIO las señales recibidas por el circuito externo para devolverlas al simulador.

Función de lectura

```
static ssize_t elevator_read(struct file *filp, char __user *buf, size_t len,
    loff_t *off) {
    int nr_bytes;
    char kbuff[BUF_LEN];

    {...}
    /* Charges the C value */
    sprintf(kbuff, "%i %i %i %i", gpio_get_value(dataRPin[3]), gpio_get_value(
        dataRPin[2]), gpio_get_value(dataRPin[1]), gpio_get_value(dataRPin[0]))
        ;
    nr_bytes = strlen(kbuff);
    /* Send data to the user space */
    if (copy_to_user(buf, kbuff, nr_bytes))
        return -EINVAL;
    {...}
    printk(KERN_INFO "Elevator: Received %s.\n", kbuff);
    return nr_bytes;
}
```

Modelos xDEVS

Como se ha mencionado previamente, se debe crear un nuevo modelo atómico *MDCosim1* que haga de enlace entre el simulador y el driver. Su labor principal es recibir los cambios en las entradas $B_3B_2B_1$ y $A_3A_2A_1$, por medio de la función δ_{ext}, para enviar al driver los nuevos valores y, posteriormente, leer del mismo el nuevo resultado, incorporándolo a través de la función λ al simulador. La función δ_{int} no tiene otra labor más que dormir al modelo a la espera de nuevos cambios en las entradas.

La descripción del modelo *MDCosim1* según el formalismo DEVS es la siguiente:

$$MDCosim1 = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

- $P \equiv \{pin14, pin3, pin5, pin15, pin2, pin6\}$
- $X = \{(i14, pin14), (i3, pin3), (i5, pin5), (i15, pin15), (i2, pin2), (i6, pin6)\}$,
  con $pin14, pin3, pin5, pin15, pin2, pin6 \in \{0, 1\}$
- $S = \{(s, \sigma, P) \mid s \in \{"active", "passive"\}, \sigma \in \mathbb{R}_0^+\}$
- $Y = \{(o10, pin10), (o13, pin13), (o1, pin1), (o4, pin4)\}$,
  con $pin10, pin13, pin1, pin4 \in \{0, 1\}$
- $\delta_{int}(phase, \sigma, P) = ("passive", \infty, \emptyset)$
- $\delta_{ext}(phase, \sigma, P, t_e, X') = ("active", 24 * 10^{-9}, F_D \leftarrow P')$
- $\lambda("active", \sigma, P) = (\{s4, s3, s2, s1\} \leftarrow F_D)$

Las funciones δ_{ext} y λ implementadas en xDEVS, las cuales interactúan con el driver, se muestran a continuación.

```
1       public void deltext(double e) {
2           if (inputHasChanged()) {
3               String cmd = "0b" + valueAtPin15 + valueAtPin3 + valueAtPin6 +
                    valueAtPin14 + valueAtPin2 + valueAtPin5;
4               driver.write(cmd);
5               super.holdIn("active", delay);
6           }else{
7               super.passivate();
8           }
9       }
10
11      public void lambda() {
12          if(super.phaseIs("active")){
13              String result = driver.read();
14              setValuesToPins(result);
15          }
16          this.oPin1.addValue(this.valueToPin1);
17          this.oPin10.addValue(this.valueToPin10);
18          this.oPin13.addValue(this.valueToPin13);
19          this.oPin4.addValue(this.valueToPin4);
20      }
```

Se necesita ahora un modelo acoplado nuevo, *RPiP1*, que contenga al modelo atómico implementado, *MDCosim1*, junto a los demás modelos de los circuitos integrados para realizar la co-simulación. Los cambios de este nuevo modelo respecto al modelo *Elevator* son los siguientes:

- Se elimina el modelo *IC74283* y sus conexiones con otros modelos.
- Se añade el modelo *MDCosim1* y se acoplan sus entradas y salidas.
- El resto de conexiones permanecen iguales.

La descripción del modelo acoplado según el formalismo DEVS es la siguiente:

$$RPiP1 = \langle X, Y, C_i, EIC, IC, EOC \rangle$$

- $X = \emptyset$
- $Y = \emptyset$
- $C_{StimulusFile} = \{stimulus\}$
  $C_{Clock} : \{clock\}$
  $C_{Constant} : \{vcc, gnd\}$
  $C_{IC74169} : \{counter\}$
  $C_{MDCosim1} : \{adderExt\}$
  $C_{IC7410} : \{nands\}$
  $C_{IC7404} : \{nots\}$
  $C_{Console} : \{(console2, console1, console0)\}$
- $EIC = EOC = \emptyset$
- $IC = \{(nots.pin2, nands.pin3), (nots.pin4, nands.pin4), (nots.pin6, nands.pin5),$
  $(nots.pin8, adderExt.i6),(nots.pin10, adderExt.i2),(nots.pin12, adderExt.i15),$
  $(nands.pin6, nands.pin1), (nands.pin6, nands.pin2),(nands.pin12, counter.pin7),$
  $(nands.pin12, counter.pin10), (adderExt.o1, nots.pin3), (adderExt.o4, nots.pin5),$
  $(adderExt.o10, counter.pin1), (adderExt.o13, nots.pin1), (counter.pin12, console2.in),$

$(counter.pin12, adderExt.i14)$, $(counter.pin13, console1.in)$, $(counter.pin13, adderExt.i3)$,
$(counter.pin14, console0.in)$, $(counter.pin14, adderExt.i5)$, $(vcc.out, counter.pin16)$,
$(vcc.out, nands.pin13)$, $(vcc.out, nands.pin14)$, $(vcc.out, nots.pin14)$,
$(gnd.out, counter.pin3)$, $(gnd.out, counter.pin4)$, $(gnd.out, counter.pin5)$,
$(gnd.out, counter.pin6)$, $(gnd.out, counter.pin8)$, $(gnd.out, nands.pin7)$,
$(gnd.out, nots.pin7)$, $(clock.oClk, counter.pin2)$, $(stimulus.sw7, counter.pin9)$,
$(stimulus.sw2, nots.pin13)$, $(stimulus.sw1, nots.pin11)$, $(stimulus.sw0, nots.pin9)$,
$(stimulus.stop, clock.stop)\}$

Resultados de la co-simulación

Utilizando el mismo fichero de estímulos que en la simulación completa en xDEVS, `Test1.txt`, la salida por pantalla es idéntica. De manera que se ha realizado la co-simulación correctamente.

```
# t = 0
Q2::0.0:0
Q1::0.0:0
Q0::0.0:0

Q2::0.5:0
Q1::0.5:0
Q0::0.5:0

# t = 1
Q2::1.5:0
Q1::1.5:0
Q0::1.5:0

# t = 4
Q2::4.5:0
Q1::4.5:0
Q0::4.5:1

# t = 5
Q2::5.5:0
Q1::5.5:1
Q0::5.5:0

# t = 6
Q2::6.5:0
Q1::6.5:1
Q0::6.5:1

# t = 7
Q2::7.5:1
Q1::7.5:0
Q0::7.5:0

# t = 10
Q2::10.5:0
Q1::10.5:1
Q0::10.5:1

# t = 11
Q2::11.5:0
```

```
Q1::11.5:1
Q0::11.5:0
```

5.4. xDEVS - 74LS283 y 74LS04

En esta parte se extraerá de la simulación el circuito integrado *74LS04*, que contiene 6 puertas NOT, y se añadirá físicamente al circuito externo. Por lo tanto, hay que adaptar las distintas partes que componen la cosimulación (comunicación, driver y módulos xDEVS) al nuevo circuito externo.

Circuito externo y comunicación

El circuito externo se amplía con el circuito integrado *74LS04*, mostrado en la Figura 5.6, que negará los bits de la entrada del piso deseado, para convertirlo a complemento-2, y negará también los tres bits menos significativos del resultado para hacer la comparación en el simulador. El diseño se puede ver en la Figura 5.7 y la abstracción de entradas y salidas en la Figura 5.8.

Al igual que en la parte anterior, las entradas y salidas del circuito son:

- $A_3A_2A_1$: Operando que guarda el piso actual.
- $B_3B_2B_1$: Operando que recibe el piso destino.
- $S_3S_2S_1$: Resultado de la resta negado, para utilizar como habilitador de la cuenta.
- UD : Indica si la cuenta será ascendente o descendente.

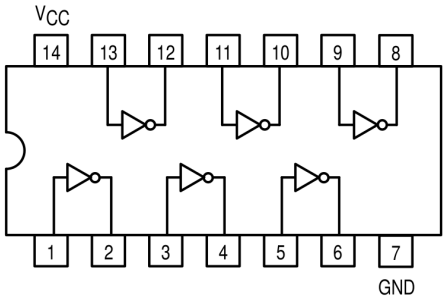

Figura 5.6: Circuito integrado 74LS04 y la posición de sus pines.

La comunicación, como en la parte anterior, será asíncrona en paralelo. Los datos $A_3A_2A_1$ y $B_3B_2B_1$ se envían al circuito externo y, tras al menos 54ns [2], se reciben UD y $S_3S_2S_1$.

La conexión entre los pines del periférico GPIO y el circuito externo se puede ver en el Cuadro 5.2 y el acabado final en la Figura 5.10 y Figura 5.11.

[2] Retardo: 54ns = 15 ns de puertas NOT + 24ns del sumador + 15ns de la puertas NOT

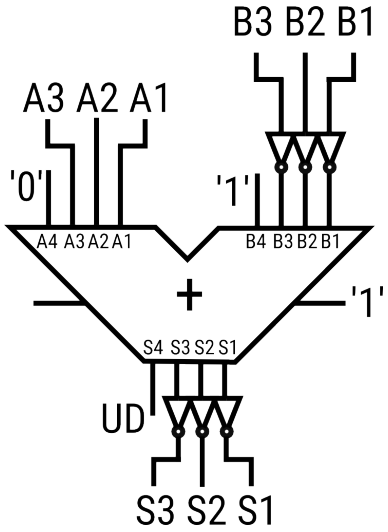

Figura 5.7: Diseño del circuito lógico externo.

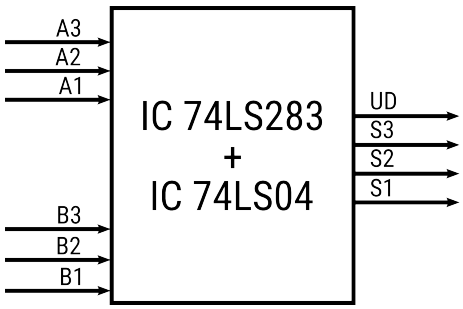

Figura 5.8: Abstracción de entradas y salidas del circuito externo.

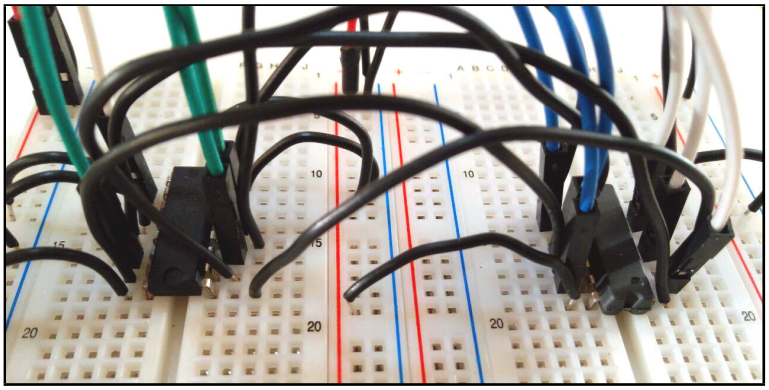

Figura 5.9: Circuito externo.

Driver

Las entradas y salidas del circuito externo son prácticamente las mismas que en la parte anterior, y la comunicación también. De forma que el driver de la parte anterior, `rpi-cosim1.c`, funciona perfectamente en esta co-simulación. La diferencia es que el circuito externo nos va a dar diferentes salidas UD y $S_3S_2S_1$ que en la parte anterior, para las mismas entradas $A_3A_2A_1$ y $B_3B_2B_1$.

SIGNAL	GPIO Pin	74LS283 Pin	74LS04 Pin
A_3	14	14	-
A_2	15	03	-
A_1	18	05	-
B_3	25	-	13
B_2	08	-	11
B_1	07	-	09
UD	12	10	-
S_3	16	-	02
S_2	20	-	04
S_1	21	-	06

Cuadro 5.2: Conexiones entre GPIO, 74LS283 y 74LS04 para las distintas señales.

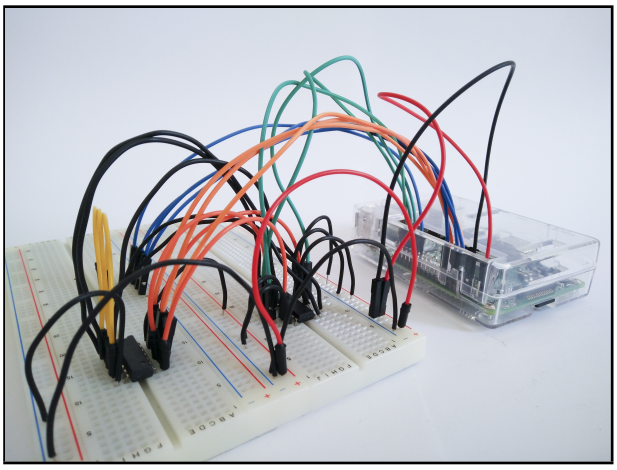

Figura 5.10: Raspberry Pi y Circuito externo - 1.

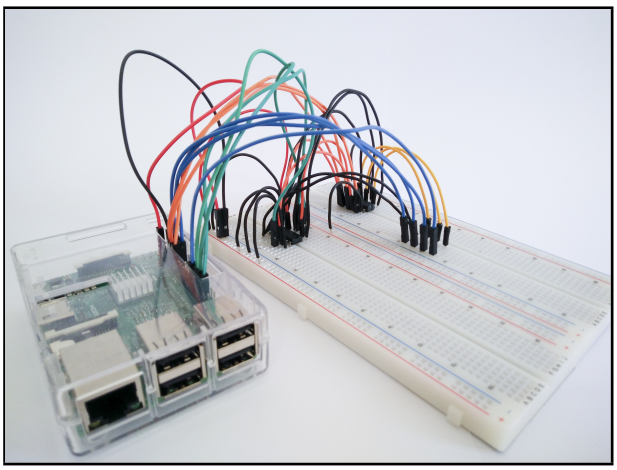

Figura 5.11: Raspberry Pi y circuito externo - 2.

Módulos xDEVS

El modelo atómico va a realizar una función muy similar a la implementada en la parte anterior, pero no se puede reutilizar el código porque los nombres de las variables hacen referen-

cia a los pines del circuito integrado *74LS283*. Entonces, se va a duplicar el modelo atómico `MDCosim1.java` en `MDCosim2.java` y se van a reestructurar los nombres de las variables, haciendo referencia a la abstracción de entradas y salidas del circuito [Figura 5.8], siguiendo el Cuadro 5.3. El resto del código será igual que en la parte anterior.

`MDCosim1.java`	`MDCosim2.java`
Pin1	S2
Pin2	B2
Pin3	A2
Pin4	S1
Pin5	A1
Pin6	B1
Pin10	UD
Pin13	S3
Pin14	A3
Pin15	B3

Cuadro 5.3: Cambio de los nombres de las variables entre los ficheros.

La descripción en el formalismo DEVS de *MDCosim2* es:

$$MDCosim2 = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

- $X = \{(iA_3, A_3), (iA_2, A_2), (iA_1, A_1), (iB_3, B_3), (iB_2, B_2), (iB_1, B_1)\}$,
  con $A_3, A_2, A_1, B_3, B_2, B_1 \in \{0, 1\}$
- $S = \{(s, \sigma, \{A_3, A_2, A_1, B_3, B_2, B_1\}) \mid s \in \{"active", "passive"\}, \sigma \in \mathbb{R}_0^+\}$
- $Y = \{(oUD, UD), (oS_3, S_3), (oS_2, S_2), (oS_1, S_1)\}$,
  con $UD, S_3, S_2, S_1 \in \{0, 1\}$
- $\delta_{int}(phase, \sigma, \{A_3, A_2, A_1, B_3, B_2, B_1\}) = ("passive", \infty, \emptyset)$
- $\delta_{ext}(phase, \sigma, \{A_3, A_2, A_1, B_3, B_2, B_1\}, t_e, \{(iA_3, A'_3), (iA_2, A'_2), (iA_1, A'_1), (iB_3, B'_3), (iB_2, B'_2), (iB_1, B'_1)\}) = ("active", 54 * 10^{-9}, F_D \leftarrow \{B'_3, B'_2, B'_1, A'_3, A'_2, A'_1\})$
- $\lambda("active", \sigma, \{A_3, A_2, A_1, B_3, B_2, B_1\}) = (\{ud, s3, s2, s1\} \leftarrow F_D)$

Partiendo del modelo acoplado de la parte anterior, *RPiP1*, en el modelo acoplado actual, *RPiP2*, se debe suprimir al modelo *IC7404* junto con todas sus conexiones y sustituir al modelo *MDCosim1* por el modelo *MDCosim2*, modificando sus conexiones para adaptarlas al nuevo circuito externo. Su descripción en el formalismo DEVS es como sigue:

$$RPiP2 = \langle X, Y, C_i, EIC, IC, EOC \rangle$$

- $X = \emptyset$
- $Y = \emptyset$
- $C_{StimulusFile} = \{stimulus\}$
  $C_{Clock} : \{clock\}$
  $C_{Constant} : \{vcc, gnd\}$
  $C_{IC74169} : \{counter\}$
  $C_{MDCosim2} : \{externCircuit\}$
  $C_{IC7410} : \{nands\}$
  $C_{Console} : \{(console2, console1, console0)\}$

- $EIC = EOC = \emptyset$
- $IC = \{(nands.pin6, nands.pin1), (nands.pin6, nands.pin2), (nands.pin12, counter.pin7), (nands.pin12, counter.pin10), (externCircuit.oUD, counter.pin1), (externCircuit.oS3, nands.pin3), (externCircuit.oS2, nands.pin4), (externCircuit.oS1, nands.pin5), (counter.pin12, console2.in), (counter.pin12, externCircuit.iA3), (counter.pin13, console1.in), (counter.pin13, externCircuit.iA2), (counter.pin14, console0.in), (counter.pin14, externCircuit.iA1), (vcc.out, counter.pin16), (vcc.out, nands.pin13), (vcc.out, nands.pin14), (gnd.out, counter.pin3), (gnd.out, counter.pin4), (gnd.out, counter.pin5), (gnd.out, counter.pin6), (gnd.out, counter.pin8), (gnd.out, nands.pin7), (clock.oClk, counter.pin2), (stimulus.sw7, counter.pin9), (stimulus.sw2, externCircuit.iB3), (stimulus.sw1, externCircuit.iB2), (stimulus.sw0, externCircuit.iB1), (stimulus.stop, clock.stop)\}$

Resultados de la co-simulación

Tras ejecutar la simulación, los resultados obtenidos son los mismos que en los casos anteriores. Por lo que la co-simulación de esta parte se ha realizado correctamente.

Salida de la co-simulación

```
# t = 0
Q2::0.0:0
Q1::0.0:0
Q0::0.0:0

Q2::0.5:0
Q1::0.5:0
Q0::0.5:0

# t = 1
Q2::1.5:0
Q1::1.5:0
Q0::1.5:0

# t = 4
Q2::4.5:0
Q1::4.5:0
Q0::4.5:1

# t = 5
Q2::5.5:0
Q1::5.5:1
Q0::5.5:0

# t = 6
Q2::6.5:0
Q1::6.5:1
Q0::6.5:1

# t = 7
Q2::7.5:1
Q1::7.5:0
```

```
Q0::7.5:0

# t = 10
Q2::10.5:0
Q1::10.5:1
Q0::10.5:1

# t = 11
Q2::11.5:0
Q1::11.5:1
Q0::11.5:0
```

5.5. xDEVS - 74LS283, 74LS04 y 74LS10

En esta parte se extraerá de la simulación el circuito integrado *74LS10* y se incorporará al circuito externo. Como consecuencia, el driver y los modelos de xDEVS sufrirán pequeños cambios.

Circuito externo y comunicación

Al circuito externo se le introduce el circuito integrado *74LS10*, cuya composición interna y posición física de los pines se puede ver en la Figura 5.12. Está compuesto por 3 puertas lógicas NAND de las cuales se utilizarán 2. Una puerta NAND valdrá para reconocer si el ascensor ha llegado a su destino, al recibir "111"de las puertas NOT, y en consecuencia deshabilitar el contador. La otra NAND va a valer para hacer la función de una puerta NOT, puesto que se necesita y no se dispone de ella.

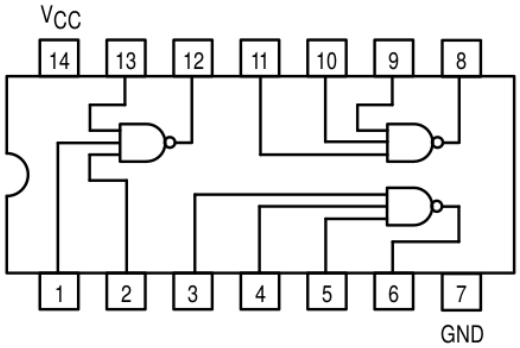

Figura 5.12: Circuito integrado 74LS10.

El diseño del circuito lógico completo se puede ver en la Figura 5.13 y la abstracción de entradas y salidas en la Figura 5.14.

Las entradas y salidas del circuito son:

- $A_3A_2A_1$: Entrada del operando que guarda el piso actual.
- $B_3B_2B_1$: Entrada del operando que guarda el piso destino.
- UD : Salida que indica si la cuenta debe ser creciente o decreciente.
- EN : Salida que indica si el contador debe funcionar (0) o debe permanecer parado (1).

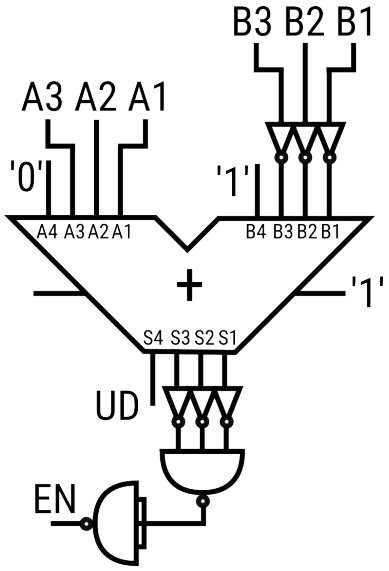

Figura 5.13: Diseño del circuito lógico externo.

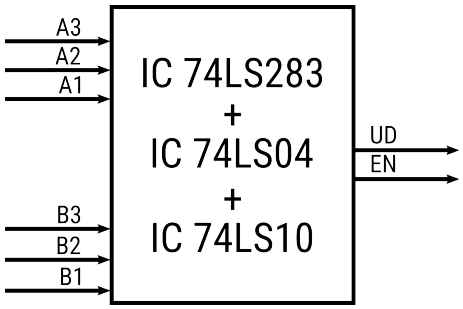

Figura 5.14: Abstracción de entradas y salidas del circuito externo.

La comunicación sigue siendo asíncrona y en paralelo. Se envían los datos $A_3A_2A_1$ y $B_3B_2B_1$ al circuito externo y tras el retardo de 84ns [3] se reciben UD y EN.

La conexión entre los pines del periférico GPIO y los pines de los diferentes circuitos integrados se refleja en el Cuadro 5.4.

SIGNAL	GPIO Pin	74LS283 Pin	74LS04 Pin	74LS10 Pin
A_3	14	14	-	-
A_2	15	03	-	-
A_1	18	05	-	-
B_3	25	-	13	-
B_2	08	-	11	-
B_1	07	-	09	-
UD	12	10	-	-
EN	16	-	-	12

Cuadro 5.4: Conexiones entre GPIO, 74LS283, 74LS04 y 74LS10 para las distintas señales.

[3] Retardo: 84ns = 15ns de NOT + 24ns de sumador + 15ns de NOT + 2 * 15ns NAND

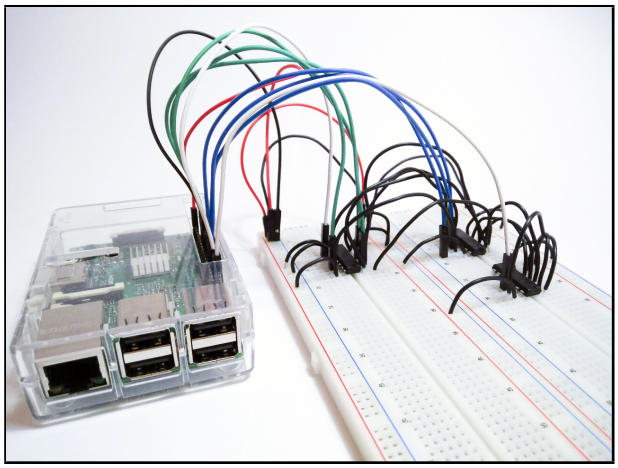

Figura 5.15: Raspberry Pi 2 con circuito externo.

Driver

El driver, `rpi-cosim3.c`, va a tener la función de escritura igual que en las partes anteriores, ya que las entradas del circuito no han cambiado. Sin embargo va a contener un leve cambio en la función de lectura, puesto que ahora no recibe cuatro bits del circuito externo, sino dos: EN y UD.

Función de lectura

```
1  static ssize_t elevator_read(struct file *filp, char __user *buf, size_t len,
       loff_t *off) {
2      int nr_bytes;
3      char kbuff[BUF_LEN];
4      {...}
5      /* Charges the C value */
6      sprintf(kbuff, "%i %i", gpio_get_value(dataRPin[1]), gpio_get_value(dataRPin
           [0])); //UD y EN
7      nr_bytes = strlen(kbuff);
8      /* Send data to the user space */
9      if (copy_to_user(buf, kbuff, nr_bytes))
10         return -EINVAL;
11     {...}
12     printk(KERN_INFO "Elevator: Received %s.\n", kbuff);
13     return nr_bytes;
14 }
```

Modelos xDEVS

Se necesita un modelo atómico nuevo, *MDCoism3*, basado en el modelo *MDCosim2*, en el cual las salidas Y, y en consecuencia la función λ, deben ser modificadas para adaptarse al nuevo circuito externo. Su descripción según el formalismo DEVS es como sigue:

$$MDCosim3 = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

- $X = \{(iA_3, A_3), (iA_2, A_2), (iA_1, A_1), (iB_3, B_3), (iB_2, B_2), (iB_1, B_1)\}$, con $A_3, A_2, A_1, B_3, B_2, B_1 \in \{0, 1\}$
- $S = \{(s, \sigma) \mid s \in \{"active", "passive"\}, \sigma \in \mathbb{R}_0^+\}$

- $Y = \{(oUD, UD), (oEN, EN)\}$, con $UD, EN \in \{0, 1\}$
- $\delta_{int}(phase, \sigma, \{A_3, A_2, A_1, B_3, B_2, B_1\}) = (\text{"}passive\text{"}, \infty, \emptyset)$
- $\delta_{ext}(phase, \sigma, \{A_3, A_2, A_1, B_3, B_2, B_1\}, t_e, \{(iA_3, A'_3), (iA_2, A'_2), (iA_1, A'_1), (iB_3, B'_3), (iB_2, B'_2), (iB_1, B'_1)\}) = (\text{"}active\text{"}, 84 * 10^{-9}, F_D \leftarrow \{B'_3, B'_2, B'_1, A'_3, A'_2, A'_1\})$
- $\lambda(\text{"}active\text{"}, \sigma, \{A_3, A_2, A_1, B_3, B_2, B_1\}) = (\{ud, en\} \leftarrow F_D)$

En xDEVS, las funciones δ_{ext} y δ_{int} se implementan igual que en `MDCosim2.java`, al ser las mismas entradas y estados, pero adaptando el tiempo del estado "$active$" en su programación en la función δ_{ext}, el cual antes era de 54ns y ahora de 84ns. La implementación en xDEVS de la función λ se muestra a continuación.

```
1      public void lambda() {
2          if(super.phaseIs("active")){
3              Integer floor = Integer.parseInt(driver.read(), 2);
4              valueToUD = (floor & (1 << 1)) != 0 ? 1 : 0;
5              valueToEN = (floor & 1) != 0 ? 1 : 0;
6          }
7          this.oUD.addValue(this.valueToUD);
8          this.oEN.addValue(this.valueToEN);
9      }
```

Partiendo del modelo previo, *RPiP2*, en el modelo acoplado necesario para esta co-simulación, *RPiP3*, se debe suprimir al modelo *IC7410* junto a todas sus conexiones. Seguidamente sustituir al modelo *MDCosim2* por el modelo *MDCosim3* y modificar los acoplamientos para adaptarlo al nuevo circuito externo. La especificación en DEVS del nuevo modelo es la siguiente:

$$RPiP3 = \langle X, Y, C_i, EIC, IC, EOC \rangle$$

- $X = \emptyset$
- $Y = \emptyset$
- $C_{StimulusFile} = \{stimulus\}$
  $C_{Clock} : \{clock\}$
  $C_{Constant} : \{vcc, gnd\}$
  $C_{IC74169} : \{counter\}$
  $C_{MDCosim3} : \{externCircuit\}$
  $C_{Console} : \{(console2, console1, console0)\}$
- $EIC = EOC = \emptyset$
- $IC = \{(externCircuit.oUD, counter.pin1), (externCircuit.oEN, counter.pin7),$
  $(externCircuit.oEN, counter.pin10), (counter.pin12, console2.in),$
  $(counter.pin12, externCircuit.iA3), (counter.pin13, console1.in),$
  $(counter.pin13, externCircuit.iA2), (counter.pin14, console0.in),$
  $(counter.pin14, externCircuit.iA1), (vcc.out, counter.pin16), (gnd.out, counter.pin3),$
  $(gnd.out, counter.pin4), (gnd.out, counter.pin5), (gnd.out, counter.pin6),$
  $(gnd.out, counter.pin8), (clock.oClk, counter.pin2), (stimulus.sw7, counter.pin9),$
  $(stimulus.sw2, externCircuit.iB3), (stimulus.sw1, externCircuit.iB2),$
  $(stimulus.sw0, externCircuit.iB1), (stimulus.stop, clock.stop)\}$

Resultados de la co-simulación

Tras ejecutar la simulación, los resultados obtenidos son los mismos que en los casos anteriores. Por lo que la co-simulación de esta parte se ha realizado correctamente.

```
# t = 0
Q2::0.0:0
Q1::0.0:0
Q0::0.0:0

Q2::0.5:0
Q1::0.5:0
Q0::0.5:0

# t = 1
Q2::1.5:0
Q1::1.5:0
Q0::1.5:0

# t = 4
Q2::4.5:0
Q1::4.5:0
Q0::4.5:1

# t = 5
Q2::5.5:0
Q1::5.5:1
Q0::5.5:0

# t = 6
Q2::6.5:0
Q1::6.5:1
Q0::6.5:1

# t = 7
Q2::7.5:1
Q1::7.5:0
Q0::7.5:0

# t = 10
Q2::10.5:0
Q1::10.5:1
Q0::10.5:1

# t = 11
Q2::11.5:0
Q1::11.5:1
Q0::11.5:0
```

5.6. xDEVS - 74LS283, 74LS04, 74LS10 y 74LS169

Esta es la última parte, en la que el ascensor estará montado completamente como la parte del hardware. El driver va a tener que interpretar más comandos para gestionar el circuito externo

y leerá, del mismo, el piso en el que se encuentra el ascensor en el momento de la lectura. Los modelos DEVS también van a modificarse, ya que el simulador sólo se va a encargar de interpretar los estímulos y generar la señal de reloj.

Circuito externo y comunicación

Al circuito externo se le va a sumar el circuito integrado *74LS169*, correspondiente al contador, cuyo circuito lógico se puede ver en la Figura 5.17 y la posición de sus pines en la Figura 5.16. Va a encargarse de guardar el piso actual del ascensor e ir incrementando o decrementando el piso, según las señales que reciba del resto de los componentes. El diseño del circuito externo completo se muestra en la Figura 5.18 y la abstracción de entradas y salidas en la Figura 5.19.

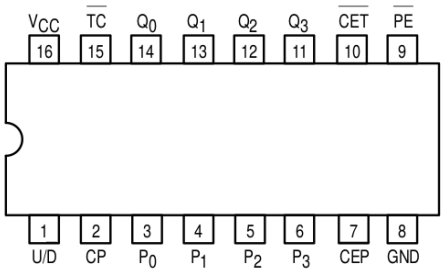

Figura 5.16: Posición física de los pines del IC 74LS169.

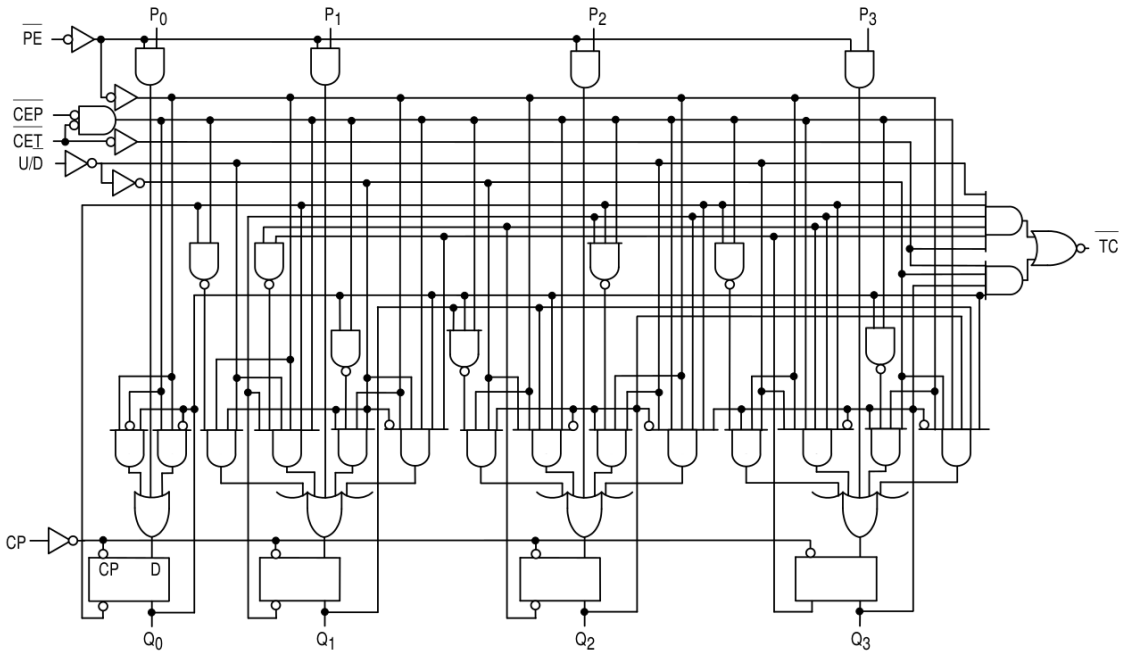

Figura 5.17: Lógica interna del circuito integrado secuencial 74LS169.

Las entradas y salidas del circuito externo son las siguientes:

- $X_3X_2X_1$: Entrada que determina el piso destino.
- Ini : Entrada que sitúa al ascensor en la planta baja, cuando su valor es 0.
- Clk : Entrada del reloj del sistema.
- $Q_3Q_2Q_1$: Salida de la planta actual del ascensor.

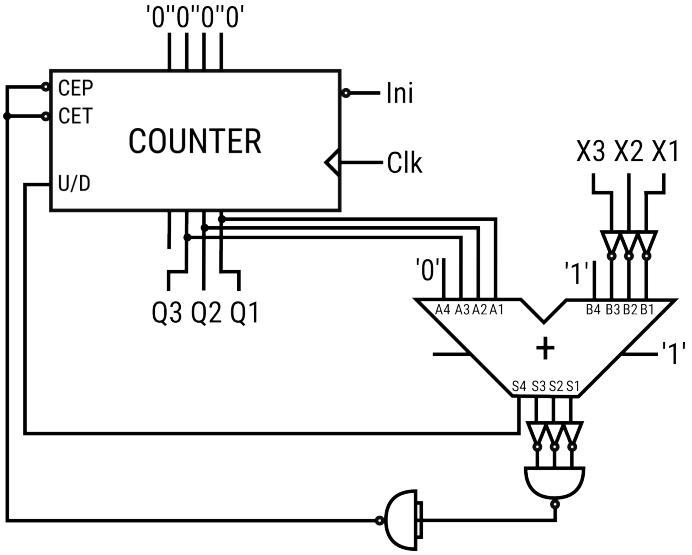

Figura 5.18: Circuito lógico del ascensor.

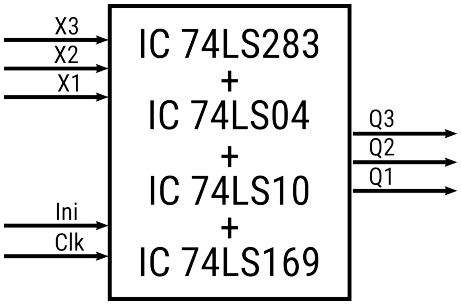

Figura 5.19: Abstracción de entradas y salidas.

Para la comunicación, que sigue siendo en paralelo y asíncrona, se envía al circuito la señal de reloj, en tiempo real, cada segundo y, con un retardo de 92ns [4], se leerá la salida del contador para detectar el cambio de planta en el simulador. Las señales $X_3X_2X_1$ e Ini se enviarán sin esperar respuesta. La conexión entre pines del circuito y GPIO se muestra en el Cuadro 5.5.

Driver

En el driver se van a modificar las operaciones de lectura y escritura. La operación de lectura va a devolver el valor del piso actual, leyendo los pines de la GPIO, y la de escritura va a admitir 3 formatos de datos:

[4]Retardo: 92ns = 24ns sumador + 15ns nots + 2*15ns nands + 23ns contador.

SIGNAL	GPIO Pin	74LS04 Pin	74LS169
X_3	25	13	-
X_2	08	11	-
X_1	07	09	-
Ini	14	-	09
Clk	15	-	02
Q_3	16	-	12
Q_2	20	-	13
Q_1	21	-	14

Cuadro 5.5: Conexiones entre GPIO, 74LS04 y 74LS169 para las distintas señales.

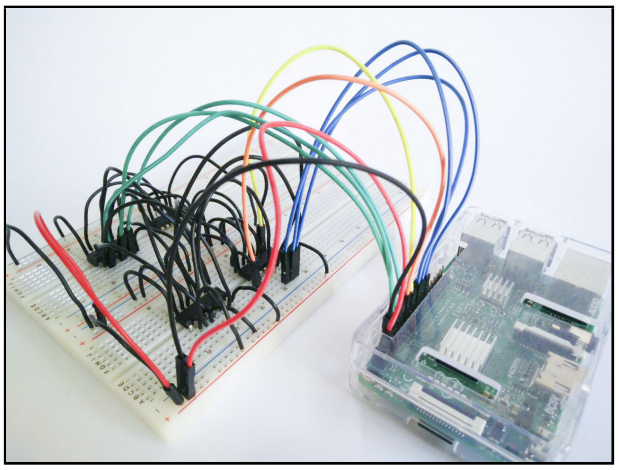

Figura 5.20: Raspberry Pi 2 con circuito externo - 1.

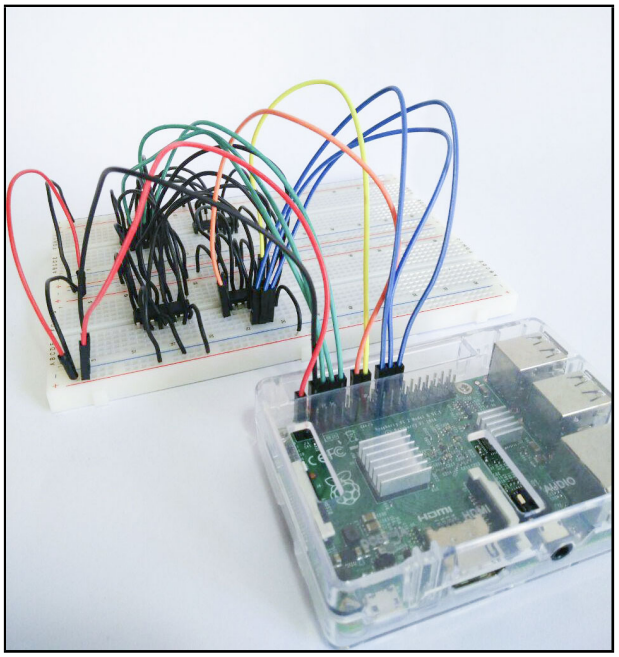

Figura 5.21: Raspberry Pi 2 con circuito externo - 2.

1 - "$0bxxx$", donde xxx correspone al piso destino en binario.

2 - "$ini\ x$", donde $x \in \{0, 1\}$ es el valor para la entrada Ini del circuito externo.

3 - "$clk\ x$", donde $x \in \{0, 1\}$ es el valor para la entrada Clk del circuito externo.

La operación de escritura debe, en primer lugar, identificar la acción recibida desde el espacio de usuario y, a continuación, enviar al pin o pines adecuados el valor o los valores correspondientes. La separación de las acciones en la escritura del driver viene dada porque desde el simulador se accederá al circuito externo en varias ocasiones con fines distintos, por lo que actualizar todas las entradas del circuito en cada operación de escritura carecería de sentido. Tambien se añade un semáforo de acceso a los pines de la GPIO, ya que el driver no lo va a usar un sólo modelo xDEVS. El código en *C* de la operación de escritura del driver se muestra a continuación.

Función de escritura

```
static ssize_t elevator_write(struct file *filp, const char __user *buf, size_t
     len, loff_t *off) {
    char kbuff[BUF_LEN];
    int value, rest, temp, i, data;
    {...}
    /* Transfer data from user to kernel space */
    if (copy_from_user( &kbuff[0], buf, len ))
        return -EFAULT;
    {...}
    /* Parse the buffer */
    if(sscanf(&kbuff[0],"0b%d", &value)){
        sendValuesToGPIO(value);
    } else if(sscanf(&kbuff[0], "clk %i", &value)){
        value = (value > 0) ? ON_ST : OFF_ST;
        if(down_interruptible(&semGpio))
            return -EINTR;
        gpio_set_value(clkPin, value);
        up(&semGpio);
    } else if(sscanf(&kbuff[0], "ini %i", &value)){
        value = (value > 0) ? ON_ST : OFF_ST;
        if(down_interruptible(&semGpio))
            return -EINTR;
        gpio_set_value(rstPin, value);
        up(&semGpio);
    } else {    /* ERROR */
        {...}
    }
    return len;
}
```

En la operación de lectura sólo hay que adaptar la salida del resultado respecto de los casos anteriores e incorporar el bloqueo del semáforo de acceso a la GPIO.

Función de lectura

```
static ssize_t elevator_read(struct file *filp, char __user *buf, size_t len,
    loff_t *off) {
    int nr_bytes;
    char kbuff[BUF_LEN];
```

```
4       {..}
5       /* Charges the C value */
6       if(down_interruptible(&semGpio))
7           return -EINTR;
8       sprintf(kbuff, "%i %i %i", gpio_get_value(dataRPin[2]), gpio_get_value(
            dataRPin[1]), gpio_get_value(dataRPin[0]));
9       up(&semGpio);
10      nr_bytes = strlen(kbuff);
11      /* Send data to the user space */
12      if (copy_to_user(buf, kbuff, nr_bytes))
13          return -EINVAL;
14      {...}
15      return nr_bytes;
16  }
```

Modelos xDEVS

En esta parte se van a desarrollar dos modelos atómicos encargados de la comunicación con el circuito. Uno de ellos, como en los casos anteriores, va a enviar al circuito externo el piso al que se desea ir y la señal de iniciación, que reinicia el circuito externo. El otro va a encargarse de enviar la señal de reloj y, tras el retardo necesario, leer el piso actual en cada momento.

El primer modelo atómico, *MDCosim4S*, va a enviar al circuito externo las señales $X_3X_2X_1$, correspondiente al piso destino, y la señal Ini, para reiniciar el ascensor. Su descripción según el formalismo DEVS es la siguiente:

$$MDCosim4S = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

- $X = \{(iX_3, X_3), (iX_2, X_2), (iX_1, X_1), (iIni, Ini)\}$, con $X_3, X_2, X_1, Ini \in \{0, 1\}$
- $S = \{(s, \sigma, \{X_3, X_2, X_1, Ini\}) \mid s \in \{"active", "passive"\}, \sigma \in \mathbb{R}_0^+\}$
- $Y = \emptyset$
- $\delta_{int}(phase, \sigma, \{X_3, X_2, X_1, Ini\}) = ("passive", \infty, \emptyset)$
- $\delta_{ext}(phase, \sigma, \{X_3, X_2, X_1, Ini\}, t_e, Ini') = ("passive", \infty, F_D \leftarrow \{Ini'\})$
  $\delta_{ext}(phase, \sigma, \{X_3, X_2, X_1, Ini\}, t_e, \{X_3', X_2', X_1'\}) = ("passive", \infty, F_D \leftarrow \{X_3', X_2', X_1'\})$
- $\lambda("active", \sigma, \{X_3, X_2, X_1, Ini\}) = \emptyset$

En este modelo la función λ no realiza ninguna acción, ya que carece de salidas Y. Su función es únicamente utilizar el driver, dentro de la función δ_{ext}, para mandar al circuito externo el piso al que se desea ir y la señal de reinicio. La implementación en xDEVS de la función δ_{ext} es como sigue.

```
1   public void deltext(double e) {
2           Integer tempIni = (iIni.isEmpty()) ? null : iIni.getSingleValue();
3           boolean activate = false;
4
5           if(tempIni != null && tempIni == 0){   //Reset counter
6               if(valueAtIni == null || valueAtIni == 1){
7                   valueAtIni = 0;
8                   driver.write("ini 0");
9               }
10          }else{
```

```
11              if((tempIni != null && tempIni == 1) && (valueAtIni == null ||
                    valueAtIni == 0)){
12                  valueAtIni = 1;
13                  driver.write("ini 1");
14              }
15          }
16          Integer tempX3 = (iX3.isEmpty()) ? null : iX3.getSingleValue();
17          Integer tempX2 = (iX2.isEmpty()) ? null : iX2.getSingleValue();
18          Integer tempX1 = (iX1.isEmpty()) ? null : iX1.getSingleValue();
19          if(tempX3 != null && !tempX3.equals(valueAtX3)){
20              activate = true;
21              valueAtX3 = tempX3;
22          }
23          if(tempX2 != null && !tempX2.equals(valueAtX2)){
24              activate = true;
25              valueAtX2 = tempX2;
26          }
27          if(tempX1 != null && !tempX1.equals(valueAtX1)){
28              activate = true;
29              valueAtX1 = tempX1;
30          }
31          if(activate && valueAtX3 != null && valueAtX2 != null && valueAtX1 !=
                null){
32              driver.write("0b" + valueAtX3 + valueAtX2 + valueAtX1);
33          }
34          super.passivate();
35      }
```

El segundo modelo atómico, *MDCosim4R*, enviará al circuito la señal de reloj y, tras 92ns [5], leerá el nuevo valor del ascensor por si hubiera cambiado. Su descripción según el formalismo DEVS se muestra a continuación.

$$MDCosim4R = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, \tau \rangle$$

- $X = \{(iClk, Clk)\}$, con $Clk \in \{0, 1\}$
- $S = \{(s, \sigma, Clk) \mid s \in \{"active", "passive"\}, \sigma \in \mathbb{R}_0^+\}$
- $Y = \{(oQ_3, Q_3), (oQ_2, Q_2), (oQ_1, Q_1)\}$, con $Q_3, Q_2, Q_1 \in \{0, 1\}$
- $\delta_{int}(phase, \sigma, Clk) = ("passive", \infty, \emptyset)$
- $\delta_{ext}(phase, \sigma, Clk, t_e, Clk') = ("active", 92 * 10^{-9}, F_D \leftarrow Clk)$
- $\lambda("active", \sigma, Clk) = (\{q3, q2, q1\} \leftarrow F_D)$

En la función δ_{ext} se envía al fichero de dispositivo del driver la señal Clk y se pasa al estado "$active$". En ese estado, en la función λ, se leerá del driver la planta actual del ascensor para detectar cambios y se enviarán por las salidas, $Q_3Q_2Q_1$, los valores leídos. La implementación en xDEVS de este modelo se muestra a continuación.

```
1       public void deltext(double e) {
2           if(!iClk.isEmpty()){
3               valueAtClk = iClk.getSingleValue();
4               driver.write("clk " + valueAtClk);
5               super.holdIn("active", delay);
```

[5] Retardo: 92ns = 23ns contador + 24 ns sumador + 15ns nots + 2*15ns nands

```
6           }
7       }
8
9       public void lambda() {
10          if(phaseIs("active")){
11              String result = driver.read();
12              Integer floor = Integer.parseInt(result, 2);
13              int mask = 1 << 2;
14              Integer tvalueToQ3 = (floor & mask) != 0 ? 1 : 0;
15              mask = 1 << 1;
16              Integer tvalueToQ2 = (floor & mask) != 0 ? 1 : 0;
17              mask = 1;
18              Integer tvalueToQ1 = (floor & mask) != 0 ? 1 : 0;
19
20              if(!tvalueToQ3.equals(valueToQ3) || !tvalueToQ2.equals(valueToQ2)
                    || !tvalueToQ1.equals(valueToQ1)){
21                  valueToQ3 = tvalueToQ3;
22                  valueToQ2 = tvalueToQ2;
23                  valueToQ1 = tvalueToQ1;
24                  this.oQ3.addValue(valueToQ3);
25                  this.oQ2.addValue(valueToQ2);
26                  this.oQ1.addValue(valueToQ1);
27              }
28          }
29      }
```

Se necesita para la simulación un modelo acoplado, *RPiP4*, que contenga a los dos atómicos descritos, al reloj del sistema y saque por pantalla los resultados por medio del modelo *Console*. La descripción del modelo acoplado en DEVS es:

$$RPiP4 = \langle X, Y, C_i, EIC, IC, EOC \rangle$$

- $X = \emptyset$
- $Y = \emptyset$
- $C_{StimulusFile} = \{stimulus\}$
  $C_{Clock} : \{clock\}$
  $C_{MDCosim4S} : \{externCircuitS\}$
  $C_{MDCosim4R} : \{externCircuitR\}$
  $C_{Console} : \{(console2, console1, console0)\}$
- $EIC = EOC = \emptyset$
- $IC = \{(externCircuitR.oQ3, console2.in), (externCircuitR.oQ2, console1.in),$
  $(externCircuitR.oQ1, console0.in), (clock.oClk, externCircuitR.iClk),$
  $(stimulus.sw7, externCircuitS.iIni), (stimulus.sw2, externCircuitS.iX3),$
  $(stimulus.sw1, externCircuitS.iX2), (stimulus.sw0, externCircuitS.iX1),$
  $(stimulus.stop, clock.stop)\}$

Resultados de la co-simulación

El resultado de la simulación es el mismo que en los casos anteriores [6]. Por lo que la co-simulación entre hardware y software de el circuito lógico de un ascensor se ha realizado correctamente.

[6] En el instante 0, las salidas $Q_2Q_1Q_0$ contendrán la planta del ascensor antes de la simulación.

Salida de la co-simulación

```
# t = 0
Q2::0.0:X
Q1::0.0:X
Q0::0.0:X

Q2::0.5:0
Q1::0.5:0
Q0::0.5:0

# t = 1
Q2::1.5:0
Q1::1.5:0
Q0::1.5:0

# t = 4
Q2::4.5:0
Q1::4.5:0
Q0::4.5:1

# t = 5
Q2::5.5:0
Q1::5.5:1
Q0::5.5:0

# t = 6
Q2::6.5:0
Q1::6.5:1
Q0::6.5:1

# t = 7
Q2::7.5:1
Q1::7.5:0
Q0::7.5:0

# t = 10
Q2::10.5:0
Q1::10.5:1
Q0::10.5:1

# t = 11
Q2::11.5:0
Q1::11.5:1
Q0::11.5:0
```

5.7. Conclusión

El diseño del circuito lógico del controlador de un ascensor ha podido ser probado y verificado por medio de simulaciones. De esta manera se han podido corregir errores en el diseño, evitando riesgos y costes innecesarios.

A continuación se han ido extrayendo del simulador los componentes del circuito externo, añadiéndolos al harware externo, pudiendo probar el funcionamiento de cada uno por partes. De

esta forma, bajo limitaciones en algunos componentes, se puede ir adaptando el diseño del circuito gradualmente. Además se pueden detectar más fácilmente piezas corruptas o quemadas, al ir sacándolas de una en una.

Finalmente, con todo el circuito montado como la parte hardware del simulador, se pueden elaborar entornos de simulación con estímulos que pongan a prueba el circuito, para poder analizar sus respuestas y comportamiento.

En conclusión, gracias a la cosimulación entre hardware y software, es posible desarrollar sistemas gradualmente, sin necesidad de descartar la simulación en etapas avanzadas del desarrollo como se suele hacer actualmente. Manteniendo un entorno simulado durante el desarrollo de los sistemas, es posible un mejor análisis y mejores evaluaciones del producto en desarrollo.

Bibliografía

[1] B. P. ZEIGLER, T.G. KIM AND H. PRAEHOFER, *"Theory of Modeling and Simulation"*, 2ª ed. New York: Academic Press, 2000.

[2] H. L. M. VANGHELUWE, *"DEVS as a common denominator for multiformalism hybrid systems modelling"*, Proceedings of the 2000 IEEE International Symposium on Computer-Aided Control System Design, pp. 129-134, 2000.

[3] JOSÉ L. RISCO Y SAURABH MITAL, *"xDEVS r20150903"* : https://docs.google.com/document/d/1_AMjvXbaUxICmLjbnIUzdrgtITLUV8p1VDKvb8OYS-Y/

[4] JOSÉ M. ANGULO USATEGUI, JAVIER GARCÍA ZUBÍA E IGNACIO ANGULO MARTÍNEZ, *"Sistemas Digitales y Tecnología de Computadores"*, 2ª ed. Madrid: Thomson, 2007.

[5] P. J: SALZMAN, M. BURIAN, O. POMERANTZ, *"The Linux Kernel Module Programming Guide"*, 2007-05-18 ver. 2.6.4.

[6] M. TERESA HORTALÁ, JAVIER LEACH Y MARIO RODRÍGUEZ, *"Matemática Discreta y Lógica Matemática"*, 3ª ed. Madrid: Editorial Complutense, 2011.

[7] EBEN UPTON, GARETH HALFACREE, *Raspberry Pi User Guide*, 1ª ed. John Wiley & Sons, 2012.

[8] RASPBERRY PI FOUNDATION, *"What is a Raspberry Pi?"* : https://www.raspberrypi.org/help/what-is-a-raspberry-pi/

[9] WIKIPEDIA, *"DEVS"* : https://en.wikipedia.org/wiki/DEVS

[10] WIKIPEDIA, *"Simulación"* : https://es.wikipedia.org/wiki/Simulaci%C3%B3n

Agradecimientos

A José Luis Risco Martín por su rápida respuesta siempre y la correcta orientación a lo largo de todo el trabajo.

A mis padres, Carlos y Amparo, y a mi otra mitad, Sonia, por el apoyo incondicional y el interés mostrado en mi trabajo.

A mis compañeros por su interés en el trabajo y ayuda, y por conseguir evadirme de vez en cuando de la materia que compone este documento.

Autorización de difusión

Autorización para la difusión del Trabajo Fin de Grado y su depósito en el Repositorio Institucional E-Prints Complutense

Los abajo firmantes, alumno y tutor del Trabajo Fin de Grado (TFG) en Ingeniería de Computadores de la facultad de Informática, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor, el Trabajo Fin de Grado (TFG) cuyos datos se detallan a continuación. Así mismo autorizan a la Universidad Complutense de Madrid a que sea depositado en acceso abierto en el repositorio institucional con el objeto de incrementar la difusión, uso e impacto del TFG en Internet y garantizar su preservación y acceso a largo plazo.

TÍTULO del TFG: **Co-simulación HW/SW en una Raspberry Pi**.

Curso académico: 2015/2016

Nombre del Alumno: **Miguel Higuera Romero**.

Tutor del TFG: **José Luis Risco Martín**, Departamento de Arquitectura de Computadores y Automática.

Firma del alumno

Firma del tutor